TAMPERE UNIVERSITY OF TECHNOLOGY

TOMMI TIKKA

A REPORTING SYSTEM FOR THE ROOT CAUSE ANALYSIS

Master of Science Thesis

Examiner: professor Hannu-Matti Järvinen
Examiner and topic approved in the Information Technology Department Council meeting on 9 February 2011

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TIKKA, TOMMI: A reporting system for the root cause analysis
Master of Science Thesis, 53 pages
May 2011
Major: Software Engineering
Examiner: Professor Hannu-Matti Järvinen
Keywords: root cause analysis, reporting system

Due to the complex software and hardware architecture and the heterogenic environment they are used in, modern mobile phones require large amounts of testing to be done before a product launch. With limited time for testing, it is impossible to find all the faults before a product launch. When a failure is detected on a device on the field, the device is delivered to the service point, where it is analyzed to determine the nature of the problem and how to fix it. The process of analyzing the faulty phones is called root cause analysis.

The root cause analysis is a process for discovering the main cause of a problem and creating recommendations on how to fix the issue in future products and current devices in the field. In a large organization, the analysis is being conducted by separate teams in multiple different geographical sites. In an environment like this it is challenging to synchronize the work of separate teams and to distribute their findings to all the sites and R&D programmes that need the information provided by the analysis.

In this thesis a reporting system is designed and implemented for the root cause analysis that provides a central location for the analysis data of multiple different sites. The main requirements set to the system are good scalability and the ability of the system to standardize the process of reporting the findings and to provide better visibility to the process to the rest of the organization.

The application designed in this thesis consists of a database and a web user interface. The application integrates to the organization's existing authentication system and it requires no additional software to be installed except a web browser. To standardize the reporting of the analysis findings, the application contains standard fields for all the analysis data, which must be filled and which ensure that the data is consistent across the organization.

All the requirements set to the application were met except the scalability requirement, which still remains unclear. The application performs well in the current situation where there are little over hundred users and data in the database from the period of six months. Still, more testing is needed with larger user base and greater data amounts in the database to determine the actual scalability of the application to multisite usage.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TIKKA, TOMMI: Perussyyanalyysin raportointijärjestelmä
Diplomityö, 53 sivua
Toukokuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastaja: Professori Hannu-Matti Järvinen
Avainsanat: perussyyanalyysi, raportointijärjestelmä

Nykyaikaiset mobiililaitteet sisältävät monimutkaisen ohjelmisto- ja laitearkkitehtuurin ja niitä käytetään vaihtelevissa ympäristöissä, joiden johdosta ne vaativat suuren määrän testausta ennen tuotejulkaisua. Rajallisen testausajan johdosta on mahdotonta löytää kaikkia vikoja ennen tuotejulkaisua. Kun vika löydetään kentällä olevasta laitteesta, se toimitetaan huoltopisteeseen, jossa laitteen vika analysoidaan ja selvitetään, miten se voidaan korjata. Tätä vika-analyysiprosessia kutsutaan perussyyanalyysiksi.

Perussyyanalyysin tarkoituksena on selvittää ongelman perimmäinen syy ja luoda suosituksia ongelman korjaamiseen tulevissa tuotteissa ja kentällä olevissa laitteissa. Suuressa laitetoimittajan organisaatiossa on monta eri ryhmää tekemässä perussyyanalyysiä eri toimipisteissä ympäri maailmaa. Tällaisessa ympäristössä on haastellista synkronoida eri ryhmien tekemä työ, jotta vältetään päällekkäisyydet ja saada prosessin tuottamaa tietoa jaetuksia kaikille sitä tarvitseville tutkimus- ja tuotekehitysohjelmille eri toimipisteisiin.

Tässä diplomityössä suunnitellaan ja toteutetaan raportointijärjestelmä perussyyanalyysiin, joka toimii keskitettynä paikkana analyysin löydösten tallentamiseen ja noutamiseen. Kyseisen järjestelmän päävaatimukset ovat hyvä skaalautuvuus, raportointiprosessin standardisointi ja prosessin läpinäkyvyyden lisääminen muualle organisaatioon.

Diplomityössä suunniteltu järjestelmä koostuu tietokannasta ja web-pohjaisesta käyttöliittymästä. Järjestelmä integroituu organisaation olemassa olevaan autentikointijärjestelmään ja sen käyttämiseen ei vaadita mitään muita asennettavia ohjelmia kuin internet-selain. Järjestelmä standardisoi analyysiprosessin pakottamalla käyttäjät täyttämään standardit kentät kaikista analyysitapauksista, jolloin jokaisesta tapauksesta löytyvät samat tiedot yhtenevässä muodossa.

Kaikki järjestelmälle asetetut vaatimukset tulivat täytetyksi paitsi vaatimus järjestelmän skaalautuvuudesta, joka on vielä selvittämättä. Tällä hetkellä järjestelmä toimii sujuvasti, kun käyttäjiä on vähän yli sata ja tietokannassa on dataa puolen vuoden ajalta. Kyseiset käyttäjä- ja datamäärät eivät vielä riitä selvittämään järjestelmän skaalautuvuutta vaan selvittämiseen vaaditaan lisää testausta suuremmilla käyttäjä- ja datamäärillä.

PREFACE

This thesis was based on a work I did at Nokia Tampere site between May 2010 and August 2010. The work and this thesis would not have been finished without the assistance and aid of several people in Nokia and in Tampere University of Technology. First, I wish to thank senior engineer Marko Meriläinen who helped me create the application and provided guidance in writing this thesis. I thank professor Hannu-Matti Järvinen from Tampere University of Technology for taking the time to examine this thesis and to guide me in the writing process. For providing assistance during the development phase and/or in the writing process, I thank M.Sc. Tomi Laine, senior specialist Petteri Ruutinen and senior manager Ahti Pylvänäinen. Lastly, I wish to thank the whole Nokia Tampere care and maintenance team for providing valuable feedback throughout the process and for providing motivating and supporting working environment.

Tampere, 5 May 2011

Tommi Tikka

CONTENTS

ABBREVIATIONS AND NOTATION

Ajax	Collection of technologies for creating dynamic web applications.
CED	Cause-and-effect diagram.
CGI	A standard defining a way to generate dynamic web pages.
CLR	A set of logic rules for the CRT diagram.
CRT	Current reality tree diagram.
CSS	Cascading Style Sheets is language for defining a visual style of a document.
DOM	Programming interface for HTML and XML-files.
ID	Interrelationship diagram.
IMEI	An unique number identifying a single mobile phone.
InnoDB	A storage engine for MySQL.
HTTP	Transport protocol for the World Wide Web.
MVC	Model, View and Controller architecture.
Octet	A sequence of eight bits.
PHP	A programming language for web development.
Primary key	A key identifying uniquely a single row in database table.
R&D	Research and development.
RCA	Root cause analysis.
RCFA	Root cause failure analysis.
Root cause analysis	A process for determining the root cause of a problem.
SQL	A language for creating database queries.
Surrogate key	An artificially created unique key for database table.
TCP	A reliable data transport protocol.
Third normal form	A criteria for determining the vulnerability of the database to logical inconsistencies.
Tuple	A single row in relational database table.
UDE	A system's undesired effects.
URI	A resource identification.
URL	A resource's location.
URN	A resource's unique name.
XMLHttpRequest	An interface for making asynchronous HTTP requests.

1 INTRODUCTION

Modern mobile phones have a very complex internal structure, consisting of various interconnected components, wires and printed circuit boards. The components operate concurrently and their outputs need to be synchronized with each other within millisecond or nanosecond timeframes. The devices also have complex software architecture with multiple operating system processes and user installed programs running at the same time. Also, these devices are used in wide array of environments and situations with varying degrees of temperature and moisture, which can affect the behavior of single components. This heterogenic and complex environment is ideal for different kind of failures or errors to appear in the devices and with limited time to test them, not all the faults are found before a new product is launched.

Failure analysis is an important part of any modern mobile phone product development process. When an error is found on a device on the field, it is analyzed to determine the nature of the problem and what caused it. If the cause of the error is fixable and there is a significant likelihood of the error reappearing in other similar devices, the error can be fixed either in the production line, so it will not appear in the devices manufactured in the future, or in the field, which usually requires the callback of each affected devices. This process, which determines the cause of the problem and the possible fix, is called root cause analysis.

In a large device provider's organization, there are usually teams conducting the root cause analysis at multiple different sites all over the world. Different sites might have different ways to store the results and report the findings. In a situation like this, it is challenging to synchronize the work of the separate teams and to distribute the findings and recommendations to all the different sites and R&D programmes, which might benefit from this information. What is needed is a standard method of storing the analysis data and reporting the findings. This can be accomplished by using a single system in all of the different sites to store the analysis data and the findings. The system can then enforce the use of standard methods to store the analysis data and report the findings. With a central location for all the data, the information may be more easily synchronized and accessed and the visibility of the process is improved thus reducing overlapping work done by teams in different sites.

The aim of this thesis is to design and implement software for reporting the root cause analysis findings and providing tools for searching the data and gathering statistics from it. The new software designed in this thesis will be based on existing root cause analysis software, which has limitations rendering it impractical for large scale

deployment. The new software will be web based and will support multiple concurrent users in a safe and efficient way.

The organization of this thesis is as follows. Chapter two introduces the main concepts, root cause analysis and web applications, needed to understand the contents of this thesis. Chapter three lists the main problems in the existing root cause analysis tool and the requirements for the new analysis software. In chapter four, the design of the new software is described with justifications for the design decisions. Chapter five describes the built software and highlights the main features. In chapter six, the results are analyzed and the encountered problems are described. The last two chapters, seven and eight, contain the ideas for further developing the software and the conclusions drawn from this work.

2 THEORETICAL BACKGROUND

This chapter describes the theory behind the root cause analysis and the theory behind dynamic web applications.

2.1 Root cause analysis

Root cause analysis (RCA), or sometimes as root cause failure analysis (RCFA), is a method or a process for determining the root cause or causes of a problem and the actions required to eliminate it (Bergman et al. 2002). Here the term root cause means the most basic reason why the problem has occurred or could occur (Wilson et al. 1993, p. 3). Another definition to the root cause is that it is the most basic cause that can be fixed and that there is no need to further divide the fixable cause (Julisch 2003, p. 14). The root cause analysis is based on the notion that the root cause of some problem is not always so obvious and eliminating the cause that at first seem to have caused the incident, may just have removed the symptom or a lower level cause and the problem may still be unresolved and appear again. Figure 2.1 depicts a problem, which appears as a symptom to the observer and has several causes at different levels. If a lower level cause is eliminated from this problem, the problem may temporarily disappear, but there is a change that the root cause will manifest itself as another problem later on (Andersen & Fagerhaug 2006, p. 13). A single problem may have many root causes (Paradies & Busch 1988, p. 479), which all are required for the problem to appear, but it is enough to eliminate just one of them to stop the problem from happening.

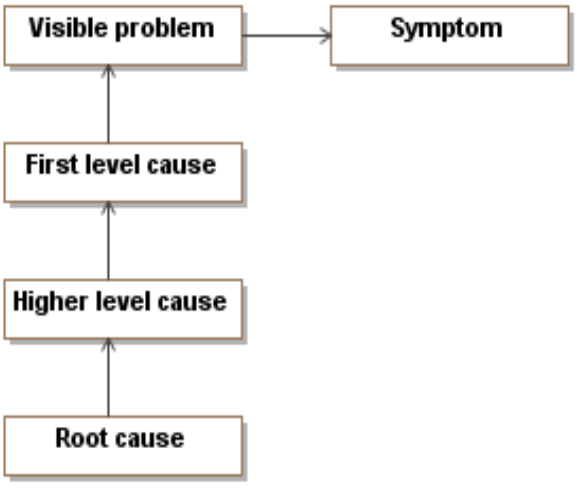

Figure 2.1 Root cause of a problem (Andersen & Fagerhaug 2006).

Root cause analysis is used in situations, where there is a problem or an unwanted behavior of a device and the problem is appearing repeatedly or there is a significant probability of it happening again. If the problem is one time only, the analysis may not be appropriate, for then eliminating the root cause or causes will have gained nothing and cost the time and resources spent on the analysis and eliminating the causes. Root cause analysis is usually performed in reactive mode (Wilson et al. 1993, p. 3), which means it is performed after a problem or a failure has occurred. The analysis is usually not applicable in problem prevention situations, where even a single problem occurrence is unacceptable. If the problem being analyzed is only hypothetical, then the whole analysis is based on guesses or estimates about the future and the results gathered from the analysis are only as accurate as the data it is based on.

2.1.1 Analysis process

There are many different variations of the root cause analysis process (Andersen & Fagerhaug 2006, p. 13). The number of process stages can vary from one organization to another and different problem areas like machinery or procedures can have separate RCA processes. A small or medium size company may for example skip some of the more formal analysis meetings and process stages, because it has fewer resources to spend in the analysis (IEEE Power Engineering Society 2007, p. 30). Figure 2.2 depicts an example of a general root cause analysis process. It consist of five stages, which deal with understanding the problem, gathering data for the analysis, identifying the root cause or causes, eliminating the problem and finally verifying the results.

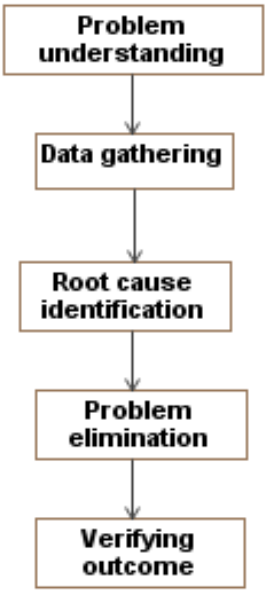

Figure 2.2 An example of a root cause analysis process.

The first step in any problem solving situation is to recognize that there is indeed a problem (Andersen & Fagerhaug 2006, p. 13), which needs to take into consideration. In a complex system like a mill or a steel factory, where there are large number of interacting people and machinery, the problem may exists a long time before the first

symptoms are noticed and the problem analyze is started. Typically the first incident notification comes as a log note, a short written message or a verbal account (Mobley 1999, p. 14). When the incident has been acknowledged, the analysis team needs to define the main problem, so the analysis is concentrated on the main issue and not some other problems that it may have caused. To define the problem, its real symptoms needs to be determined and the limit that bound the event need to establish (Mobley 1999, p. 14). When enough information is obtained to define the problem, it needs to be classified (Mobley 1999, p. 14), so an appropriate analysis process, that suits the current problem, can be selected. Common problem classifications include equipment failure, operating performance, safety and regulatory compliance (Mobley 1999, p. 16).

The next step in the root cause analysis process is to gather additional data for the root cause identification. The analysis is based on the gathered data, so it must be reliable and valid, so the results gathered from the analysis are usable. The data can be found from the incident reports, interviewing the people present at the incident, collecting physical evidence (Mobley 1999, p. 20) or from any other suitable source like log files. The data gathering takes time and it may be longest step in the analysis (Rooney & Vanden Heuvel 2004, p. 48).

When enough data is gathered from the problem, it is analyzed to determine the root cause or causes of the problem. There are many different methods and tools for root cause analysis which the more popular ones are described below. The analysis usually consists of charting the causes of the event to a diagram, which can then be used to clarify the sequence of events so the root cause can be found.

When the root cause of the event has been identified, the next step is to generate recommendations for preventing the recurrence of the problem (Rooney & Vanden Heuvel 2004, p. 49). The recommendations need to be achievable (Rooney & Vanden Heuvel 2004, p. 49) in the organization that implements them and their cost need to be less than the gains received from implementing them so they can be realistically considered. The corrective actions should also provide permanent protection from the problem, but sometimes due to financial constraints a temporary solution may be considered (Mobley 1999, p. 48). After the corrective actions list has been generated, a cost-benefit analysis can be used to select the best solution that fits the current situation and the organization (Mobley 1999, p. 48).

After the corrective actions are implemented it needs to be verified that the likelihood of the problem reappearing is lower and the application or equipment is working properly. The verifying is usually done with tests (Mobley 1999, p. 57). When the RCA process is over, the results can be documented for later reviewing.

2.1.2 Root cause identification tools

There a various tools that can be used in different stages of root cause analysis. In literature, the three most common root cause identification tools are the cause-and-effect

diagram (CED), the interrelationship diagram (ID) and the current reality tree (CRT) (Doggett 2004).

Cause-and-effect diagram

The purpose of the cause-and-effect diagram, or sometimes as fishbone diagram (Mobley 1999, p. 10), is to sort the potential causes of a problem and to organize the causal relationships (Doggett 2005). Also, the construction of the diagram promotes discussion and enables sharing information about a process or problem (Doggett 2005). The diagram (Figure 2.3) consists of the problem, the trunk of the tree, and the categories of causes affecting it, the branches of the tree. The number of categories is not set, but there are typically four categories: human, machine, material and method (Mobley 1999, p. 9). Within every category, the detailed causal factor as listed as twigs of the branch (Doggett 2005, p.35). The process of constructing the diagram is as follows (Ishikawa 1982, according to Doggett 2005). First the problem to be controlled or improved is decided. Second, the problem description is written to the right side of the diagram and an arrow is drawn to point to it. Next, the main factors that may have caused the problem are drawn as branches to the main arrow. Detailed factors are then drawn as twigs to each major factor. Detailed factors may also have more detailed factors drawn as twigs to them. Finally, it must be ensured that all the factors contributing to the problem are included in the diagram. The diagram only lists all the causes of a problem, it does not give the root cause of the problem. It is the responsibility of the creator of the diagram to study each cause and to determine, if it is the root cause. The diagram also does not contain the sequence of events that lead to the problem (Mobley 1999, p. 9), which makes it more difficult to determine the importance of a single cause to the overall problem.

The cause-and-effect diagram has few variations, which have different structures and can be used in different situations. The dispersion CED diagram uses groups of probable causes of the problem as the branches of the tree and the twigs are the reasons why variation occurs in the problem (Doggett 2005, p. 36). The advantage of the dispersion CED is that it relates causes to effects and provides a framework for brainstorming (Milosevic 2003, p. 468). The process classification CED divides the main arrow to process steps, which each have its individual branches and twigs. This adds the sequence of events to the diagram, which makes it easier to understand the effect of a single cause to the problem. The cause enumeration CED organizes all the possible causes of the problem according to their relationship to the problem and to each other (Doggett 2005, p. 36). The resulting diagram then contains a thorough collection of causes (Doggett 2005, p. 36).

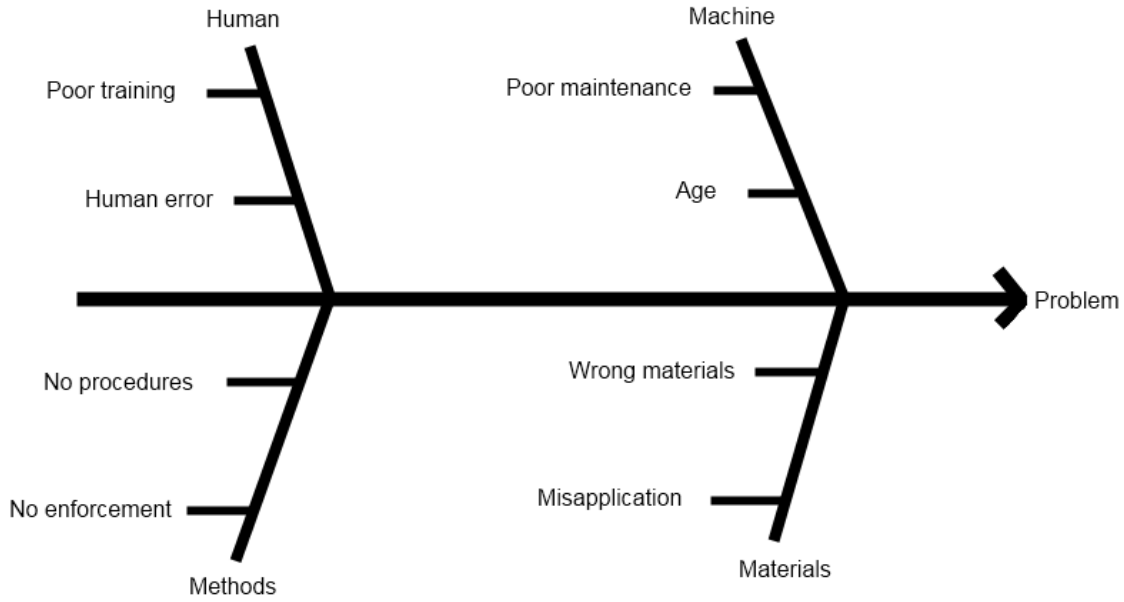

Figure 2.3 An example of cause and effect diagram (Mobley 1999).

Interrelationship diagram

The interrelationship diagram, or sometimes as relations diagram, clarifies the intertwined causal relationships of a problem so that an appropriate solution can be found (Doggett 2005, p. 37). The diagram also helps to identify the key drives or outcomes, which can be used to come up with effective solution to the problem (Base 2008). The creation of the diagram encourages the participants to think in multiple directions to discover critical issues about the problem (Doggett 2005, p. 37), which otherwise might be hard to discover using linear thinking. The diagram (Figure 2.4) is made of boxes and arrows, where each box represents a single causal factor and an arrow represents a causal relationship. The direction of the arrow describes the direction of the causal relationship: it points to the result and starts at the cause (Doggett 2005, p. 37). Next to each box is calculated the number of incoming arrows and the number of outgoing arrows. The boxes with more outgoing arrows than incoming arrows are the causes and the boxes with more incoming arrows are the effects. The diagram can also be drawn as a matrix, where the causal factors are in the columns and again in the rows. The matrix cells then describe the strength of the relationship between the factor in the column and the factor in the row. The last column is reserved for counting the total strength of the relationships of the factors in the corresponding row.

The process of creating an interrelationship diagram is as follows (Mizuno 1988, according to Doggett 2005). First the information is collected from multiple sources. Then the causal factors are named using concise phrases or sentences. After the group has reached consensus on the names, the diagram is drawn. Finally the diagram is drawn multiple times to identify and separate the critical issues. The interrelationship diagram is usually used to further examine causes and effects, which might be recorded previously in a cause-and-effect diagram (Base 2008).

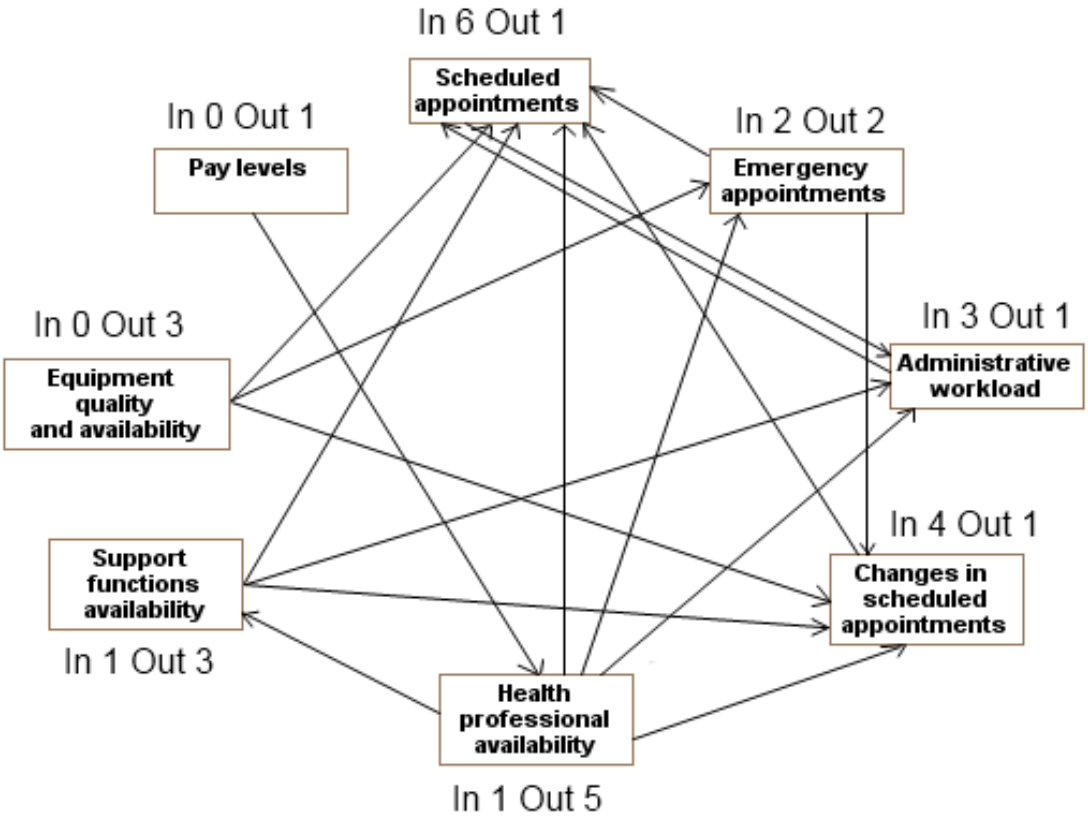

Figure 2.4 An example of interrelationship diagram (Doggett 2005, p. 38).

Current reality tree

The current reality tree helps to find links between system's undesired effects (UDEs) (Doggett 2005, p.39), which are the visible symptoms of the problem the user can see, so that the root cause of the problem can be discovered. The tree depicts the current state of the system and shows the most probable chain of cause and effect, given a specific set of circumstances (Mabin et al. 2001, p. 172). The tree (Figure 2.5) consists of entities, which are square surrounded statements about an idea, cause or an effect (Doggett 2005, p.39). The entities are linked with arrows, which imply a sufficient relationship between the entities (Doggett 2005, p. 39), meaning that the cause can create the effect. If an effect requires multiple causes simultaneously to exist, an oval is placed on the tree and the arrows are drawn from the causes to go over the oval to the effect. The oval means that all the causes are required to create the effect and if one of them is missing, the effect will not appear. The tree may also contain loops, which amplify positively or negatively some effect (Mabin et al. 2001, p. 172). The loops can be distinguished in the tree by arrows going downwards from an effect to a cause (Doggett 2005, p. 39).

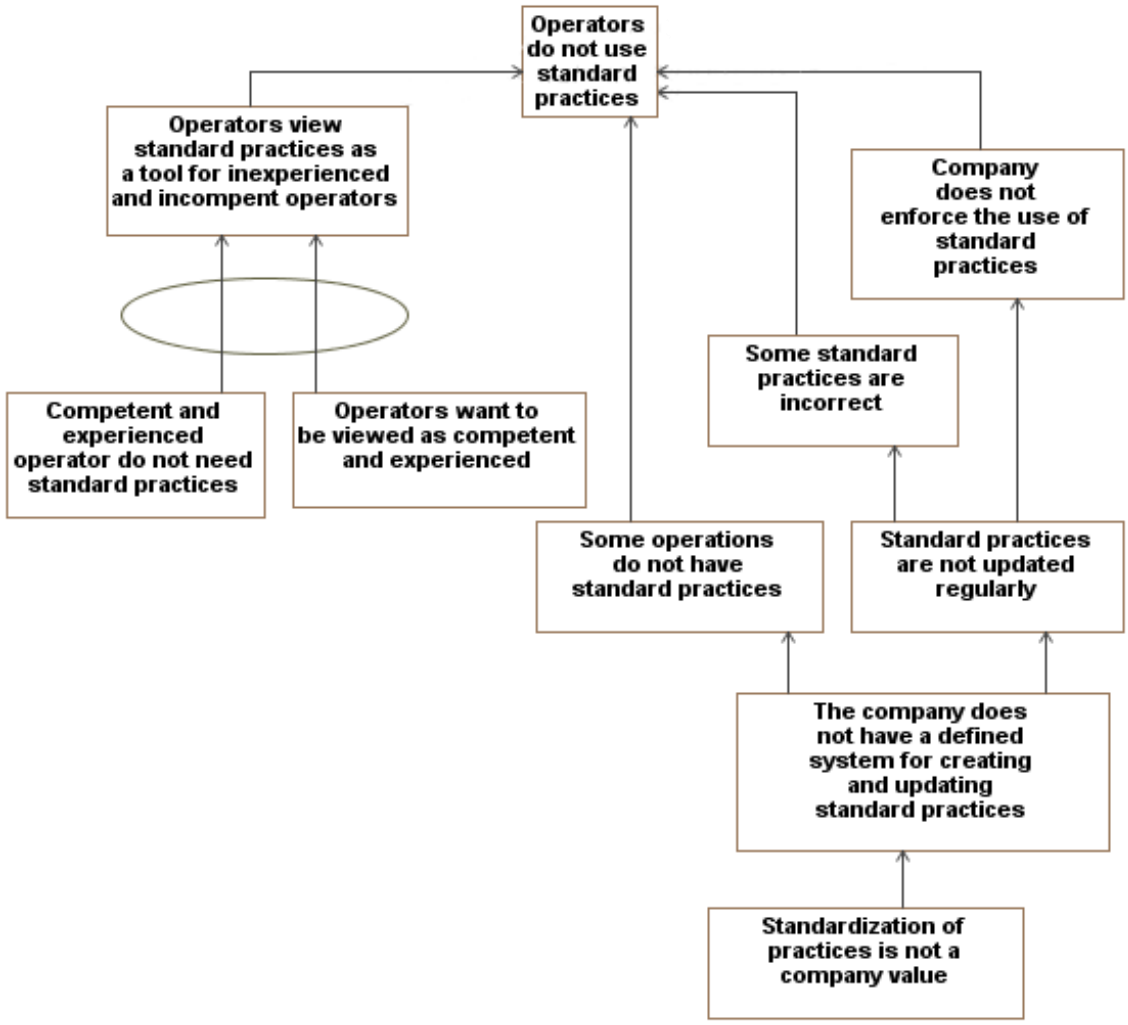

Figure 2.5 An example of current reality tree (Doggett 2005, p. 40).

The current reality tree is constructed top-down, starting from the visible symptom and going downwards by postulating the likely causes (Mabin et al 2001, p. 172) that could have caused the symptom. The created relationships are tested with a set of logic rules called categories of legitimate reservation (CLR), which ensure rigor in the CRT process by working as the criteria for verifying, validating and agreeing upon the relationships (Doggett 2005, p. 41). The procedure of constructing the tree is as follows (Cox & Spencer 1998 according to Doggett 2005). First the UDEs related to the problem are listed. The UDEs are then tested for clarity and causal relationships are looked for between any two UDEs. The relationships can be discovered by using ”if-then” statements with the UDEs, like if a valve is closed then the water will not flow. If a relationship is found, then it must be determined, which UDE is the cause and which is the effect. Finally, the relationship is tested with the CLR. The process is continued until all the UDEs are connected. If an effect requires multiple causes to exist, the causes are connected with the oval. While connecting the UDEs, the relationships can be strengthened by using words like some, few, many, frequently and sometimes.

2.2 Dynamic web applications

The World Wide Web contained originally only static content like text pages and images, which were identical to each user viewing the content. Later the technology advanced to the point it was possible to create dynamic content, which means the web page is created at the time it is requested and the content is customized to suit the user and the current usage context. Dynamic content creation enables the programmer to create applications to the web, which behave like any normal desktop application. Transferring applications to the web provides the benefits of better accessibility, maintainability and visibility. Web based application can be used in any place with an internet connection, even while moving with mobile devices. Maintaining the application is easier, when there is only one instance of the code running in the central server, which all the users use. If the application requires visibility, today's pervasive internet is the best place to provide a service, which can be seen and accessed all around the world.

The content in the World Wide Web is transferred with the HTTP protocol, which is also the protocol every web application needs to use to deliver the HTML pages to the client. The HTTP protocol uses the URI system to identify and locate the transferred resources. To make dynamic web content, a script needs to be run either in the server or in the client that creates the HTML page. PHP programming language can be used to create scripts that are run in the server, when a HTTP request is received. Ajax is collection of technologies that can be used to create scripts that are run in the client side.

2.2.1 Uniform Resource Identifiers

The Uniform resource identifier (URI) is a string of characters used to identify an abstract or physical resource (Berners-Lee et al. 1998). The resource can be anything, which has an identity (Berners-Lee et al. 1998), like a Word document, an image or a service. The string of characters the URI is made of can represent the resources current location in the network, called URL (Uniform Resource Location), or its unique name, called URN (Uniform Resource Name). The difference between URL and URN is the time the identity is valid (Berners-Lee et al. 1998). The URL is only valid while the resource stays in the same location, if it is moved, the URL changes. The URN however stays the same even if the resource's location changes. The URNs are not currently widely used (Shklar & Rosen 2003, p. 31) and in many cases the URI is used to mean specifically the URL address.

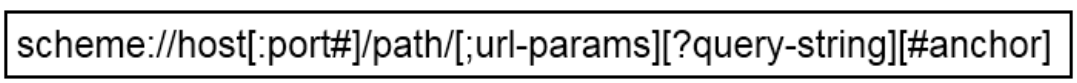

Figure 2.6 The structure of the URL address (Shklar & Rosen 2003, p. 31).

Figure 2.6 shows the general structure of the URL address. The URL contains the following components (Shklar & Rosen 2003, p.31):

- *scheme* – designates the protocol used to form the connection with the server at the given URL address. It is usually http for web browsing or ftp for file transfer.
- *host* – the IP address or the hostname of the server.
- *port* – designates the port number of the server to which the connection is established. The value is optional, the default is port number 80 for WWW browsing.
- *path* – the file system path to the resource. Can be relative, meaning the location is relative to the current locations, or absolute.
- *url-params* – contains optional parameters for the URL address. Not commonly used.
- *query-string* – contains optional parameters for the request. Is usually used with the HTTP protocol's GET method requests.
- *anchor* – optional reference to a location in the requested web page.

2.2.2 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) is an application-level data transfer protocol for distributed hypermedia systems (Fielding et al. 1999). It defines the method for transporting messages between two separate network end points identified by the URI.

HTTP is a part of the TCP/IP protocol suite and it is used in the World Wide Web (WWW) as the protocol for transmitting HTML pages and messages from the servers to the clients. HTTP usually uses the TCP (Transmission Control Protocol) for data delivery, which is a transport-level protocol for reliable two way data transfer, but it can be set to work on top of any reliable transport protocol (Berners-Lee et al. 1998). If the TCP is used, the HTTP usually uses the port number 80. HTTP is based on a request/response protocol (Berners-Lee et al. 1998), where the client sends a request message for a resource to the server, which in turn sends a response message containing the resource. The messages usually contain text in ASCII format, although other formats can be used as well (Mogul 1995, p. 299). The communication between the client and the server is rarely direct (Shklar & Rosen 2003, p. 34), there is usually devices between them, like proxies, gateways and tunnels (Berners-Lee et al. 1998), which forward the messages towards the target. The devices between the end points may read the messages, for example a firewall may check the message for viruses or worms, and even alter them, like a translating proxy changing the resources language before passing it on. If the requested resource contains many separate parts, like a web page containing images, the HTTP protocol can use persistent connection, which was added to the HTTP protocol version 1.1 (Berners-Lee et al. 1998). If the persistent connection is enabled, the connection between the client and the server is not closed every time a

resource has been transmitted and thus the negotiating process need to be done only once, when the connection is established.

HTTP is a stateless protocol (Shklar & Rosen 2003, p. 34). The word state, or sometimes as session, means the location in the sequence of commands or requests, where the interaction between the client and the server currently is (Shklar & Rosen 2003, p. 34). For example, in a web store, the user's shopping cart is the state, which needs to be maintained when the user moves one page to the next until he reaches the checkout page. The actual state data consist of name and value pairs, like *UserId = 299933392*. The state is usually maintained in the server's memory or in the file system, in protocols, which supports state, like FTP or SMTP. In HTTP protocol, the server does not need to maintain a state for the connection, which makes the protocol simpler and uses fewer resources on the server, but it makes it harder to build applications on top of the protocol.

HTTP has two message types, a request and a response message. The general structure of a message consist of a header section, one empty line and the actual body of the message (Shklar & Rosen 2003, p. 35), which is optional. The header section contains information the receiver needs to understand the message, like the message type, and it may also contain information about the message body, such as the content type, encoding or length. Each header field consists of the attribute name followed by a colon and the value of that attribute (Fielding et al. 1999). The order of the header fields is not important (Fielding et al. 1999). Figure 2.7 shows the general structure of a request message. All request messages start with the request line, which include the request method, the URI of the requested resource and the version number of the HTTP-protocol (Shklar & Rosen 2003, p. 35). After the request line there may be additional header fields, which usually contain information about the request and the client (Fielding et al. 1999), like the preferred encoding and language.

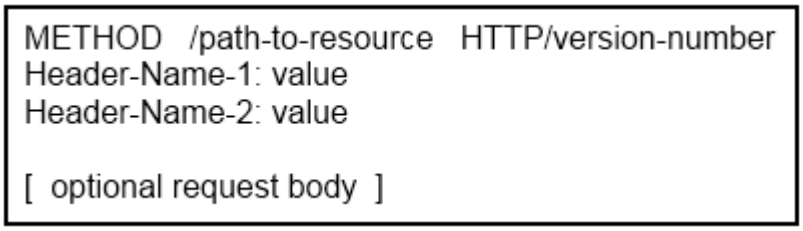

Figure 2.7 The general structure of a request message (Shklar & Rosen 2003, p. 35).

Figure 2.8 show an example of a request message. When the client inputs an URL *http://en.somesite.org/directory/page.html* to the web browsers address line, the browser sends the request message to the server with the URI *en.somesite.org*. The message requests a resource */directory/page.html* via GET request method, which is the default method for retrieving HTML pages in World Wide Web. The message contains two additional header fields, Accept and Accept-Charset, which define what kind of a response message the web browser expects the server to respond with. In this case, the

response message is expected to contain a HTML page in its body section, with the ISO-8859-1 encoding.

Figure 2.8 An example of a request message.

When a HTTP-server receives a request message, it decodes the message, locates the requested resource and creates a response message containing the resource or an error code indicating a missing resource. Figure 2.9 shows the general structure of a response message. A response message starts with a status line, consisting of the HTTP protocol version number followed by a numeric status code and its textual description (Fielding et al 1999). The status code is a three digit integer and its optional human readable description, which tells the client either that the request has been fulfilled successfully, or that the client needs to perform a specific action, which can be further parameterized with additional header fields (Shklar & Rosen 2003, p. 42). The status codes have been divided into five classes and the first number of the code is used to indicate the class. The last two digits are used to indicate the specific status code inside the class. In HTTP protocol version 1.1, the five status code categories are (Fielding et al. 1999):

- *1xx* – Informational. The request has been received and the process continues.
- *2xx* – Success. The requested has been successfully received, understood and it has been accepted.
- *3xx* – Redirection. Additional action needs to be performed in order to complete the request.
- *4xx* – Client error. The request message contains bad syntax or the request cannot be fulfilled.
- *5xx* – Server error. The server failed to fulfill the valid request.

The status line is followed by optional response header fields and entity fields, which can be used to pass additional information about the response and the requested resource (Fielding et al 1999). The response body is optional, it is used to transfer the resource to the client.

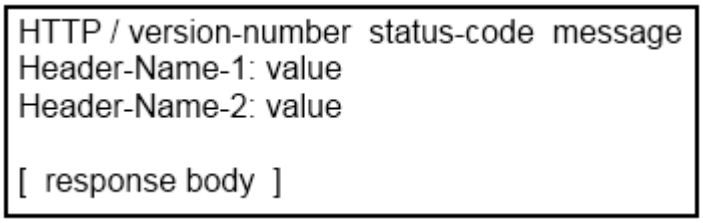

Figure 2.9 The general structure of a response message (Shklar & Rosen 2003, p. 36).

Figure 2.10 shows an example of a response message. Here the status code indicates that the request has been fulfilled successfully and the resource has been found and delivered with the message. The response contains additional header fields which indicate, that the message contains a HTML page and its length is 9012 octets. The body section contains the actual requested resource.

```
HTTP / 1.1  200  OK
Content-Type:  text/html
Content-Length:  9012

<html>
  <head>
    <title>Title</title>
  </head>
  <body>
  ...
  </body>
</html>
```

Figure 2.10 An example of a response message.

The HTTP protocol version 1.1 defines eight request methods: CONNECT, DELETE, GET, HEAD, OPTIONS, POST, PUT and TRACE (Fielding et al. 1999). The methods define the action needed to perform in order to complete the request. Of the eight methods, GET and HEAD are called safe methods (Fielding et al. 1999), which means they only perform resource retrieval and do not take any action on the resource itself. This makes them safe to be used in any situation and if they do cause some side-effects, the user cannot be held accountable for them (Fielding et al. 1999). The methods that can be repeatedly called, with no additional side effects, are called idempotent methods and they include the methods GET, HEAD, PUT, DELETE, OPTIONS and TRACE (Fielding et al. 1999). Of all methods in the HTTP protocol, the most commonly used are the GET, HEAD and POST (Shklar & Rosen 2003, p. 37).

The CONNECT method name is a special case of the HTTP methods. It is reserved for use with a proxy that can change it to a tunnel dynamically (Fielding et al. 1999).

The DELETE method can be used to request the server to delete a resource. The resource to be deleted is identified by a URI in the request. The server responses to the DELETE request with the status code describing if the resource was deleted successfully.

When the user enters a URL in the browser or clicks a hyperlink, the browser uses GET method to retrieve the web page (Shklar & Rosen 2003, p. 38). GET method is used to retrieve a resource without any side effects on the server. The requested resource is identified by the URI header field (Fielding et al. 1999), which can be a relative or absolute address. GET request message contains no body and the only

required header field in HTTP version 1.1 is the Host-field used with virtual hosting (Shklar & Rosen 2003, p. 38).

```
GET  /index.html?name=John&age=20  HTTP / 1.1
Host: www.app.net
```

Figure 2.11 A GET request message with parameters.

With a GET request message additional parameters can be given that specify, for example, the selected category when viewing book listings. The parameters are placed in the resource's URI after a question mark. Figure 2.11 shows an example of a GET request message that contains two parameters, name and age. With the parameters in the URL of the web page, GET queries can be bookmarked with the web browser like any other web address. This can be used to store the state of the web application with only bookmarking the URL address, which cannot be done with the POST method.

The HEAD method is identical with the GET method except the server sends only the headers fields in response to the request and the body section is omitted. HEAD method is used to request information from the server, like for example, the modification date of the requested resource. This can be used to support client caching, where the client stores the retrieved web page locally and upon re-entry to the same page, asks the server if the resource has changed since it last was requested. If there is no change to the resource, the local version can be used, otherwise a new GET request is made. HEAD method can also be used with change-tracking systems, for testing and debugging new applications or for learning the server's capabilities. (Shklar & Rosen 2003, p. 41-43.)

Information can be requested from the server, like its capabilities or requirements associated with a resource (Fielding et al. 1999), without initiating any action, with the OPTIONS method. The request may contain a URI specifying the resource the information concerns with. The server sends a response containing the requested information in the header fields.

The POST method can be used to deliver data to the server like a message to a bulletin board or form data to the data handling process (Fielding et al. 1999). Unlike in the GET method, POST methods parameters are in the message's body section and they do not show in the URL. POST method can therefore be used to hide the data transfer to server from the user.

```
POST  /index.html  HTTP / 1.1
Host: www.app.net

name=John&age=20
```

Figure 2.12 A POST request message with parameters.

Figure 2.12 shows an example of a POST request message with contains two parameters, name and age. The request is identical with the one in Figure 2.11 except

the parameters in Figure 2.12 are not visible in the URL and the query cannot be bookmarked.

The PUT method can be used to store a new resource in the server. The request must contain the entity to be stored in the message's body section. The difference between the POST and the PUT method is the meaning of the request-URI (Fielding et al. 1999). In the POST method the supplied URI specifies the handler of the entity, whereas in the PUT method the URI identifies the entity (Fielding et al. 1999). If the supplied URI already identifies a resource in the server, the entity in the request must be considered an update version of the said resource and it should replace the original. The server responses to the PUT request with a status code indicating if the request was completed successfully or not.

The TRACE method is used to diagnose the request chain between the client and the server. All the proxies between the client and the server will write their address in the header fields and the final recipient of the request will send the request back to the sender. When the client receives the reply message, it contains all the addresses of all the devices between the client and the server and it also contains the original message in the body section. This way the client can see what kind of data is received in the server end and which route the request take to reach the server.

2.2.3 Server-side scripting and PHP

When a server receives a HTTP request, it locates the requested web page and, if it contains only static content, returns it immediately. If the web page contains dynamic content, then the server needs to perform some actions and as a result of those actions, a web page with the content is created, which the server returns to the sender. There are many different techniques for creating dynamic content on the server side, which can be divided to categories on the basis of their approach to web development (Shklar & Rosen 2003, p. 246). The categories are: programmatic approach, template approach, hybrid approach, and frameworks (Shklar & Rosen 2003, p. 246). The programmatic approach category covers techniques, where the web application's source code consist mostly of code written in Perl, Python or some other high level programming language like Java (Shklar & Rosen 2003, p. 246). In these cases, the language offers methods or environment variables that can be used to retrieve information about the HTTP request such as the URL header or parameters (Shklar & Rosen 2003, p. 247). The actual HTML page generation is usually done by printing methods offered by the language. CGI (Common Gateway Interface) is one of the prevalent methods for creating dynamic content (Thiemann 2002, p. 1) in programmatic approach category. With CGI, a programming language not designed for web development, can be used to create dynamic web pages.

In template approach, the source object, from which the final HTML page is generated, consist mostly of HTML-code, which can have embedded code constructs

(Shklar & Rosen 2003, p. 249) that create the dynamic part of the page. Special tags are usually used to indicate the part containing programming code. Unlike in the programmatic approach, the focus of the template approach is in the formatting of the web page, not in the programming logic (Shklar & Rosen 2003, p. 249). This can make it easier to design the outlook of the page, when the source object resembles more the final result. An example of a well known technique using template approach is Cold Fusion, which provides tags for including external resources, conditional processing, iterative result presentation and data access (Shklar & Rosen 2003, p. 250).

Hybrid approach is a mixture of programmatic approach and template approach (Shklar & Rosen 2003, p. 254). In hybrid approach, the HTML code contains blocks of code, which have the programmatic power of a normal programming language. The source object, in hybrid approach, has the page formatting benefits of templates, but it can also contain the logic of programmatic approach. An example of hybrid approach is the PHP-language.

PHP (PHP Hypertext Preprocessor) is programming language designed for creating lightweight web applications (Trent et al. 2008). It is dynamically typed, interpreted language, which support object-oriented programming from version three onwards. When a server with PHP interpreter receives a request for a resource with suffix .php or php3, it locates the resource and sends it to the PHP interpreter. The interpreter then executes the code inside the PHP tags and replaces the code blogs with the print statements. The result is a HTML page with PHP code replaced by the output of the print commands.

2.2.4 Asynchronous JavaScript and XML

Ajax (Asynchronous JavaScript and XML) is a collection of technologies aimed at making web pages more responsive, meaning faster loading times and more dynamic content. In a non-Ajax web application every change to the content of the page, like loading additional images as the result of the user selecting different image category, results in a complete reloading of the entire page (Paulson 2005, p. 14) while the user's web browser is unresponsive and waiting for the request to complete. With Ajax, a web application can behave more like desktop application and reload only the changed portion of the page and while reloading, the application can continue interacting with the user (Paulson 2005, p. 15).

Ajax consists of the following technologies: dynamic HTML, XML, DOM and XMLHttpRequest. Ajax uses HTML, CSS and JavaScript to create dynamic web pages. JavaScript running at the client end can change the contents of the HTML page or CSS-style sheet without any action required from the server. This enables fast changing HTML pages even on a slow internet connection, because the HTML page needs to be loaded only once. In Ajax all the data transfer between the client and the server are XML-encoded (Paulson 2005, p. 15). XML (Extensible Markup Language) is a markup

metalanguage, which can be used to define languages dealing with structured data (Paulson 2005, p. 15). With XML, the client and server can use different internal formats for the same data, but can still share it with each other by using a common XML-format. When the XML-structured data arrives to the client or the server, it can be transformed with XSLT (Extensible Stylesheet Language Transformations) (Garrett 2005, p. 1) to the internal format the receiver uses.

DOM (Domain Object Model) is a programming interface that can be used to modify HTML- and XML-documents (Paulson 2005, p. 15). DOM presents the document as a tree like structure containing objects (Asleson & Schutta 2006, p. 39). Each object can contain attributes like background-color or height and each object knows its parents and siblings. With DOM, it is possible, for example, to change the color of text by changing the document object's text color attribute or to delete all the list items by going through all its sibling objects and deleting them.

XMLHttpRequest is an application interface for making HTTP request asynchronously without the user's web browser becoming unresponsive while waiting for the server's response (Paulson 2005, p. 15). The process of using XMLHttpRequest is as follows (Asleson & Schutta 2006). First the XMLHttpRequest is created as a JavaScript object or ActiveX component. The created XMLHttpRequest instance contains methods for making HTTP request either with GET or POST and it is given a pointer to the callback function, which is called when the state of the request changes. When the request is send, the server begins to process the request and the XMLHttpRequest instance releases the control back to the caller. When the HTTP response message has arrived the XMLHttpRequest instance calls the callback function, which then updates the page or does whatever action is needed to perform to the received data.

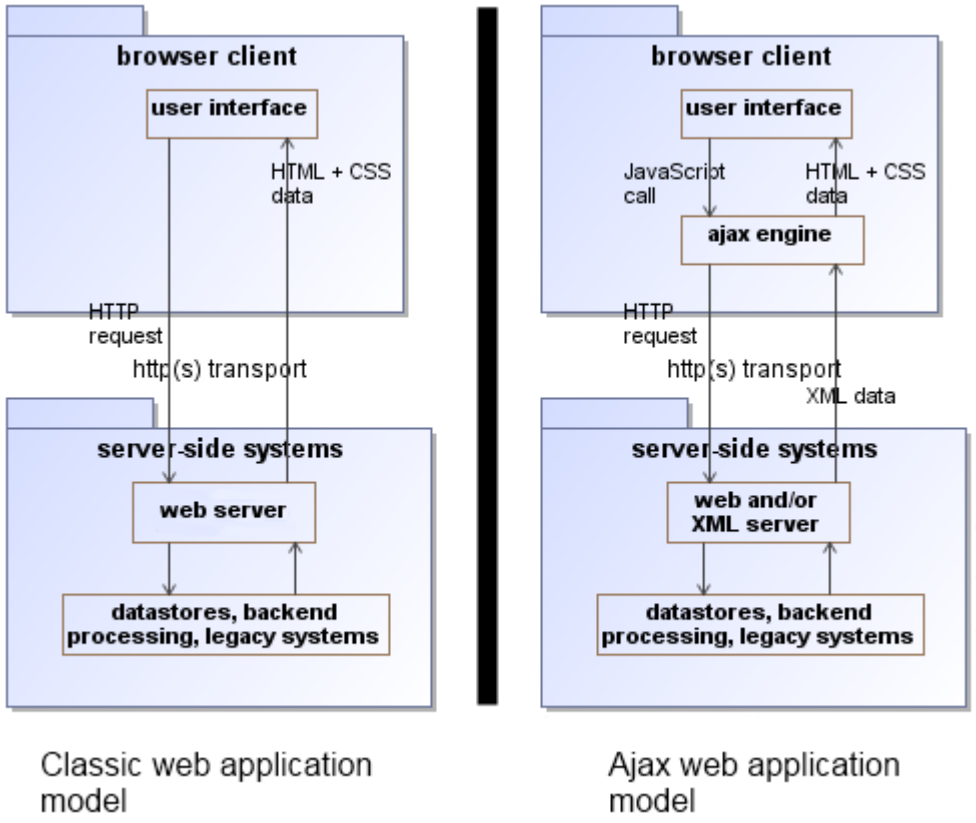

Figure 2.13 The difference between classic and Ajax web application models (Garrett 2005, p. 2).

Figure 2.13 shows the classic web application model and the Ajax web application model side by side. In the classic web application, most user actions trigger a HTTP request to web server (Garrett 2005, p. 1). The server then handles the request, retrieves the resource and sends a response containing a new HTML page, while the user interface is locked in waiting for the response message. When the browser receives the response, the currently open page is replaced with the HTML page in the response. The main difference with the classic model and Ajax model is the Ajax engine, which handles the communication with the server and rendering the user interface (Garrett 2005, p. 2). In the Ajax model, the user actions trigger JavaScript method calls to the Ajax engine, requesting for the update of the page. If the update needs action from the server, the engine then sends a normal HTTP request and releases the control back to the browser, making the application again responsive for the user's actions. If the page update is a minor one like validating input fields, the engine can handle it own without HTTP communication (Garrett 2005, p. 2). When the server responds with the new XML data, the Ajax engine updates the user interface, but only the part which needs to change.

3 REQUIREMENTS

The application designed in this thesis is based on an existing root cause analysis reporting application. The existing application has shortcomings, which renders it unsuitable for multisite deployment. The requirement specification for the new application was done before the work done in this thesis began. Some of the requirements in the specification were changed during planning of the new application and some were changed during the development phase. The sections 3.2 – 3.5 contain the requirements that were valid, when the development phase began.

3.1 Problems with the existing system

The existing system was created in 2005 with Microsoft Access UI Builder and it used Microsoft Access database for data storage. The purpose of the system is to keep record of analyze data gathered from mobile phone hardware and software analyses and to provide tools for tracking the status of the ongoing analyses and generating various reports from the stored data. The architecture of the system is client-server based. The client has the executable software, containing the UI and business logic components, and the server has the data storage. The server is usually located in a local R&D site. The system is designed for small group local usage with around ten simultaneous users per database.

The main features of the system are the adding and modifying mobile phones, analysis data, assigning tasks to specific phones, generating reports from the data and managing the supporting data like mobile phone categories and user access levels. Each analysis case has a fixed set of fields, where the analysis data could be given in ASCII format. Tasks like “return the phone to sender” can be added to cases. They can be emailed to the person assigned to the task. Reporting tool supports Microsoft Word and Excel formats and the reports can be printed straight from the system.

The system requires that each user has a local copy of the client software. When a new version of the system is deployed, each user needs to copy the updated version to the local hard drive and remove the old version of the system. When there are many users using the system, there is a risk that multiple versions of the client software are using the same database. If the data structures have been modified in the update, there is a risk of corrupting or breaking the database.

The client software uses a local R&D site’s database located in a server’s network drive as the data storage. The network drive, where the database is located, must be visible in the client’s operating system and the location of the database must be

configured in the client software. If the database is moved to a different network drive, every client needs to map the new drive to the operating system and the new location to the client software. Due to performance problems, a single central database cannot support the usage from multiple sites, located in different countries. This means that each site has a local copy of the data common to multiple sites and each site also has the data that is used only in the current site. If there is change to the data common to multiple sites, every site needs to synchronize its data with the other sites. If the sites are far apart from each other and located in different time zones, the data synchronization can be an expensive and difficult operation.

The existing system has searching and filtering capabilities, but they have usability issues making them difficult to use. Due to this, the less experienced users tend to avoid using them, which makes the tool less efficient to use and increases the work load of the user. The system has predefined input fields for information concerning a single case, but what is missing is the ability to add user defined metadata like pictures or files to cases. This means that the metadata needs to be maintained in another system and there is no link between a case in the existing system and its metadata in another location, like a network drive.

3.2 Database

3.2.1 Database distribution

The new system's database will be divided to three separate partitions: global, local and archive. The global partition contains the information of all the sites, while the local partition contains the information of the local site only. The archive partition is for maintaining old cases, so the information contained in them can be searched but not modified. The purpose of the partition is to increase the performance of the system by limiting the operations on local data to the local partition and to only use the global partition when it is necessary. The distributed database provides a single point, where the analysis information is stored and which therefore reduces the overlapping work done in different sites and the need to replicate data between multiple database systems. The system also reduces the possibility that multiple sites have different versions of the same data due to too infrequent synchronizing. By keeping the analysis data of multiple sites in single distributed database, the system also enhances visibility between groups working in different sites and reduces the possibility that the same problem will be analyzed many times by different groups.

3.2.2 Data migration

The new system needs a tool for data migration from the old database to new one. The existing system has data from a long time period that is valuable in the root cause analysis. To be available for reference purposes this data needs to be transferred to the new system.

3.3 Feature

3.3.1 Case locking

Only one user can edit a specific analysis case at time. If other users try to edit the same case, they are notified that the case is already locked for someone else. It should be possible to view a case while someone else is editing it. To prevent a user from locking a case indefinitely, the system needs to have a timer, which opens the case when a certain amount of time has passed.

3.3.2 Messaging tool

The application needs to have a messaging tool, which informs the analyst, by email, of cases stayed open for too long. This reduces the possibility that a single case can be forgotten, when an analyst has many simultaneous cases. The tool also informs the analyst, in charge of a case, when the case is being edited by other users. The messaging tool can also be used as a communication channel between users, who do not know each other, but still work with the same case.

3.3.3 Attachments

An analysis case must have the option to insert attachments to it. Any metadata, like log files, can be added to a case and stored in the database. Local metadata are replicated to global partition enabling teams in another sites can also access them. By storing the case related metadata with the analysis data, the analyst can quickly access all the case related files. Then there is no need to use external application to look, for example, the devices log files.

3.3.4 History data

There shall be a record kept of all the editing done in the system. When a user modifies a case, the time, date and the target of the modification is recorded to the database for later viewing. The history data can be later used to track the changes of an analysis and build a timeline of the process. The data can also be used to identify malicious users.

3.3.5 URL links

There should be an option to add URL links to cases. The links can be used to refer to separate data, located in different web-based systems.

3.4 General

3.4.1 Web application

The new system shall be a web-based application. The user should not have to require installing any additional software, besides a web browser, to use the system. If the application is updated, the new version will automatically be in use, when the user opens the URL address of the application. The web based application also enables roaming use, where the user can use the application with a mobile phone or a wireless laptop in different locations and even while in motion.

3.4.2 Standardize RCA reporting

The new system needs to provide a standard method for reporting the root cause analysis findings. The system needs to guarantee that all the data added to the system is in the same format and that each analysis case contains all the required information.

3.4.3 Scale to multisite environment

The system needs to work in multisite environment. The database queries need to be efficient so they run reasonably fast with many concurrent users and large amount of data in the database.

3.5 Usability

3.5.1 Filtering and search

The new system should have extensive searching and filtering capabilities. They need to be simple enough for temporary users to benefit from them, but they also need to be effective enough that they satisfy the search and filtering requirements of seasoned users. The search should be divided to two different search types, index search and full text search. The index search will only search the value of a single attribute, but it does it efficiently. The full text search need not to be as efficient, but it must be able to search for a match in every possible attribute. Both search types should run the search in three database partitions, which are global, local and archive.

3.5.2 User help

Help feature is required for the new and temporary user. It must provide instructions how to use the system. It may also contain tips for using the system more effectively. The help material needs a search feature, which can be used to quickly search for a solution to a specific problem.

4 DESIGN

The system designed in this thesis consists of a database holding the analysis data and a web user interface, which is used to access the data. The most important requirement dictating the structure of the database is efficiency. The database needs to store large amounts of data, while still being able to provide efficient queries to the data. To reach that goal, the database queries have been designed to use only simple table joins and table indexes are used whenever it is possible.

The application design is based on MVC (Model, View and Controller) architecture. In MVC architecture, the responsibility of the data storage, business logic and the user interface is divided to separate modules. MVC architecture leverages the adaptability of the application by limiting the changes concerning either the data storage, control logic or user interface to their modules.

4.1 Structure of the analysis information

The basic unit of information in the application is a case. It represents single mobile phone failure information with analysis data. Each case has a unique identification number which is used to identify the case from other cases in the system. Unique identification number is generated automatically when a new case is added to the system. A case has a state, which is either open or closed. Cases with open state are ongoing and case data is editable. After the analysis is completed, the case state is turned to close and the data is locked. A single case contains the basic information of the phone (Table 4.1), analysis data (Table 4.2), comments, links to metadata and its modification history.

Table 4.1 Phone information.

Attribute	*Description*
Case ID	Identifies a case.
IMEI	Identifies a mobile phone.
HW version	Hardware version number.
SW version	Software version number.
Main symptom	Description of the main symptom.
Secondary symptom	Description of the possible secondary symptom.

Table 4.2 Analysis information.

Attribute	*Description*
Status	The case status.
General problem area	The general problem location.
Detailed problem area	The more detailed problem location.
Root cause	Description of the root cause.
Action	Description of the main symptom.

Figure 4.1 shows the relationships between a case, its analysis, comments and metadata. Each case can have any number of metadata attached to it, like all the files inside the field system of the phone. Although the same metadata, like some library file, can in reality exist in multiple phones, this system does not support adding the same metadata to multiple cases. A single case can have any number of comments, which represent the textual description of the analysis process. Each case also has exactly one analysis.

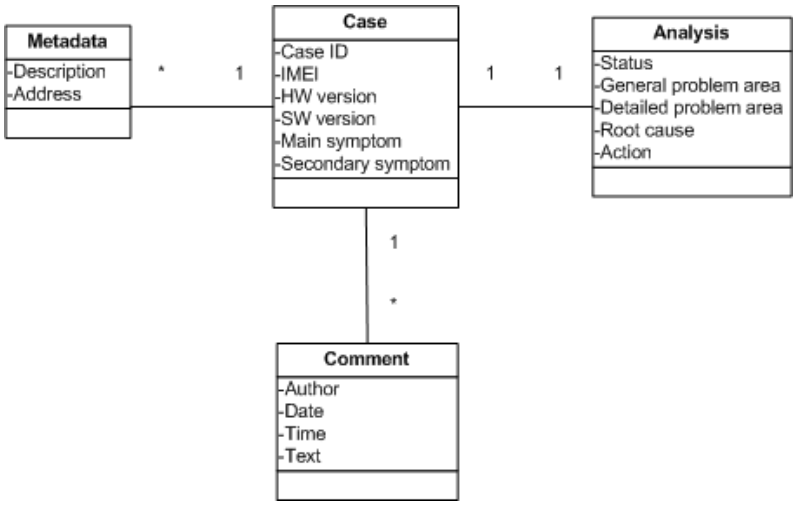

Figure 4.1 Relationships between a case and its data.

An analysis data (Table 4.2) represents the results of the root cause analysis performed on the phone. It contains the location of the problem given in two separate fields, *general problem area* and *detailed problem area*. This division to two areas separated by the abstraction level is done to provide more possibilities to search for cases. With two location fields, the search may be limited to a general problem area or a more detailed problem area may be provided. The most important information in the analysis data is *root cause*. It provides the description of the cause, which is a prerequisite for the problem and which must be fixed in order to prevent the recurrence of the problem. The analysis data also contains the recommended action to fix the issue. *Action* field contains the description of the first step, which must be taken to fix the issue. This might be for example a remainder to inform the manufacturing department to

change some component or it might be just text informing that no action is required, because the incident will not reappear.

The performed analysis is written as comments to a case. The reason for the written comments is to improve the visibility of the analysis. A person outside the analysis process may track the analysis by reading the comments, which describe what was done or discovered in the analysis as it progressed. By using written comments, the analyst in the case may be changed and the new analyst can quickly get fully informed of the case by reading the comments. A new comment is added to the case when the analyst performs something or discovers something that is important to the analysis.

4.2 Database architecture

The database stores all the analysis data generated in the root cause analysis process. In addition, all the supporting data that the application uses are also stored in the database. The database consists of the following tables (Figure 4.2): *Analysis*, *Attachment*, *Case_history*, *Comment*, *Country*, *Phone*, *Problem_area*, *Product*, *Production_data*, *Round*, *Sub_area*, *Sym_code*, *Sym_group* and *User*. Most of the tables use automatically generated surrogate key as their primary key with a few exceptions that use the value of one of the columns as the primary key. By using automatically generated surrogate keys the risk of two tuples having the same primary key is minimized and key comparison can be made efficiently compared to natural keys, where each of them may consist of the concatenated values of multiple columns.

The database is normalized to the third normal form. By normalizing the tables, the size of a single table is smaller and it contains only relevant data. Normalizing to the third form reduces the dependencies between tables so the insert, update and remove operations need to touch smaller number of tables and are therefore performed more efficiently. The normalization also provides easier data integrity management by removing the redundant data.

The type of the database tables is InnoDB. It supports transactions and it adds foreign key support to the database tables. By using transactions several queries can be linked together which means all or none of them is completed. This application uses transactions in such operations which require several database queries to complete at the same time, like new case insertion to the system.

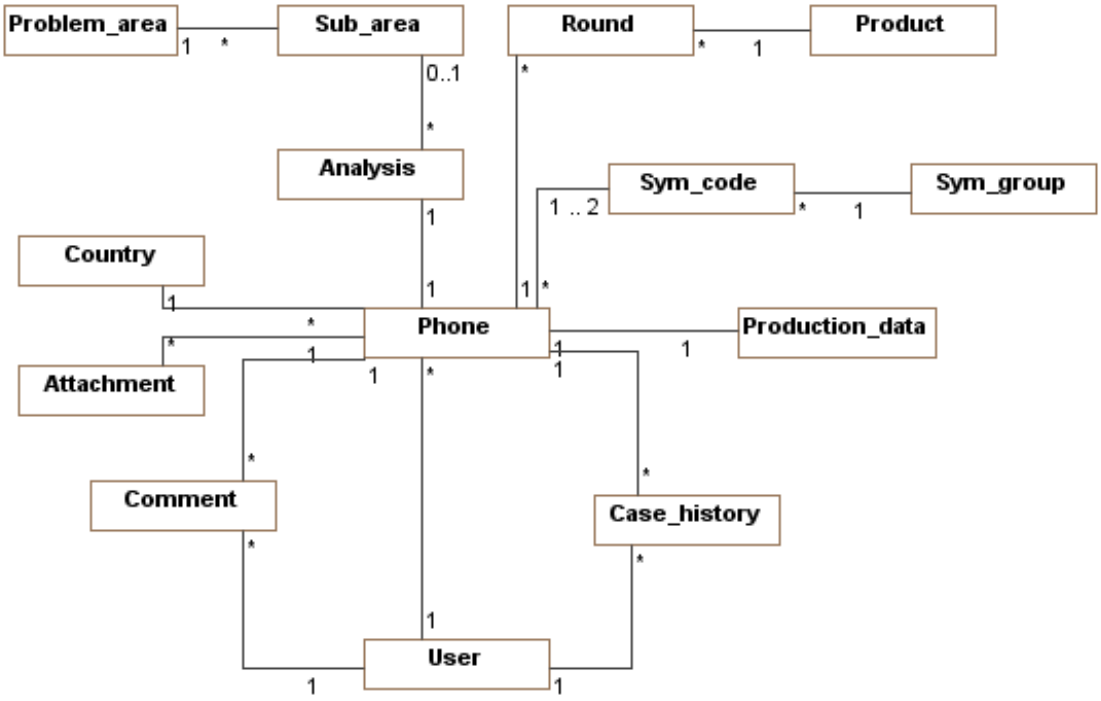

Figure 4.2 The relationships of the tables in the database.

Figure 4.2 shows the relationships of the database tables. The *Phone* table is the central table in the database architecture. The reason for this is that most of the database queries involve searching some case specific information with the phone's id value. Fewer calculations are needed and the queries can be completed faster by keeping the table joins at minimum.

The *Phone* table is connected to the *User* table with three separate links. The three links represent the three separate roles the user plays in relation to the phone. The first role is the analyzer, who writes analysis comments. This is represented by the link going through the comment table. This table is used to store the comment text and the timestamp it was made. The direct link between the *Phone* table and the *User* table represents the holder of the phone. The third link is used to store the history data. *Case_history* table stores all the editing a user does. The data can be later used to create a timetable of all the modifications of a case.

All the findings made in a single root cause analysis case are stored in the *Analysis* table. This table is connected to the *Sub_area* table, which in turn is connected to the *Problem_area* table. These tables contain the detailed category and the more general category, where the discovered problem is located. "User interface" is an example of a general problem category and "Input text field" is an example of a more detailed category.

The list of symptoms of a case is stored in the *Sym_code* and the *Sym_group* tables. Like the problem areas, the symptoms are divided to two different categories as well. The *Sym_code* table stores the detailed symptom and *Sym_group* table stores the general

category. “Power” is an example of general category and “Phone does not power on” is an example of a more detailed symptom.

The *Round* table keeps a record of the source where a sample was collected. This information can be used to identify the product of the case, so there is no need to link the *Product* table with the *Phone* table.

4.3 Application architecture

The architecture of the application is MVC. The application consists of three modules, which are *Model*, *View* and *Controller*.

The *Model* module is responsible for the data of the application. The classes in the module are the only classes with a direct access to the database. The classes in the other modules need to use the services of the model in order to access the data in the database. The *Model* module keeps the applications data stored in the database, offers methods for retrieving and updating the data, validates the data before it is inserted to the database and keeps the data up to date. The module also hides the database from the rest of the application, which reduce the dependencies between the application and the database. This makes the application more portable and easier to maintain. If the database is relocated or the database management system is changed, only the *Model* module needs to be modified, the rest of the application continue to use the application through the model’s unchanged public interface.

The *View* module contains the user interface of the application. The module consists of classes, which each represents a single screen in the application. The views contain no business logic and the data they require to display to the user is provided to them by the classes in the *Controller* module. All the user input received by the views is forwarded to the controller classes for processing. The *View* module hides the visual appearance of the application from the other modules. By limiting the user interface modifications to a single module, the portability of the application is enhanced.

The *Controller* module contains the business logic of the application. The classes in the module create the views, provide data to them from the model module and handle all the user input and any other higher functionality. The module acts as a median between the *View* module and the *Model* module. It delivers data from the *Model* to the application’s screens. It also receives the user input from the views, validates it and sends it to the *Model*, which stores it in the database.

5 IMPLEMENTATION

This chapter lists some of the key features of the finished application.

5.1 Technologies used

The core classes of the application have been written with PHP programming language. The language is immensely popular among web based applications meaning there is a good chance that the language is maintained and supported for long time in the future. The PHP programming language is also easy to learn, which makes application maintenance and further development easier.

The application was build by using CodeIgniter framework. The framework offers helper functions and libraries which speed up the process of creating web based MVC applications.

The user interface of the application has been written with JavaScript to make it more dynamic and to make it respond more quickly to user input. The JavaScript code uses JQuery library to decrease the amount of code lines and to make the code more readable. JQuery is a JavaScript library that offers methods for common Ajax interaction like document traversing or event handling. By using the JQuery library, common Ajax operations like a HTML element selection can be made browser independent way and with only one method call.

The application uses several widgets from the JQueryUI library. The JQueryUI library is a collection of animations, effects and widgets that can be used in the user interface of a web application. The widgets in the library are highly customizable and they can be themed with CSS styles.

The application uses the Free PDF library to generate PDF-reports. The library offers methods for creating PDF-documents with a support for many different font families and encodings. All the item lists have been created with the help of DataTables library. It is a JavaScript-based Microsoft Excel-like table for web applications.

5.2 Main views

The main views of the application are *Add case, My open cases, Meeting view, Search* and *Admin*. *Add case* view (Figure 5.1) is used to add new analysis case to the system. The important fields are mandatory, which makes sure each case has the minimum amount of information to begin the analysis. Mandatory fields also standardize the reporting of phone information across different sites. *Symptom selection* fields (see

Table 4.1) are filtered according to the text written inside them. This makes it easier to find appropriate symptoms for the phone.

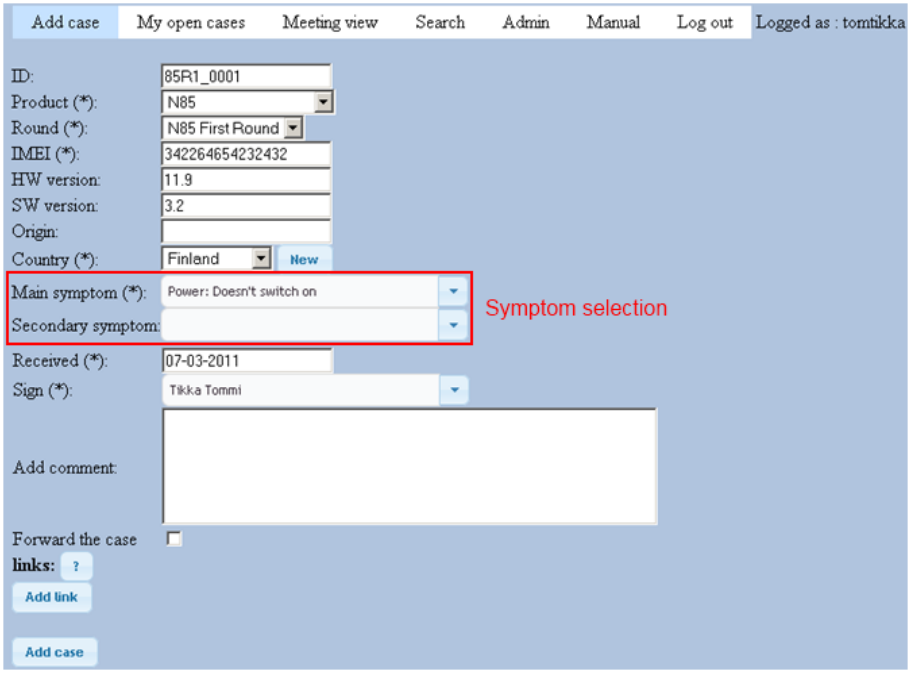

Figure 5.1 Add case view.

My open cases and *Meeting view* (Figure 5.2) contain *case table*. *My open cases* list the open cases of the current user and *Meeting view* lists all the cases in the system. *Case table* can be filtered in real time with *table filter*. This improves the usability by making it easier to find specific cases from a long table. *Meeting view* also contains *case filter*, which filters the cases that are retrieved from the server and inserted to the table (see section 5.9 for the difference between the two filters).

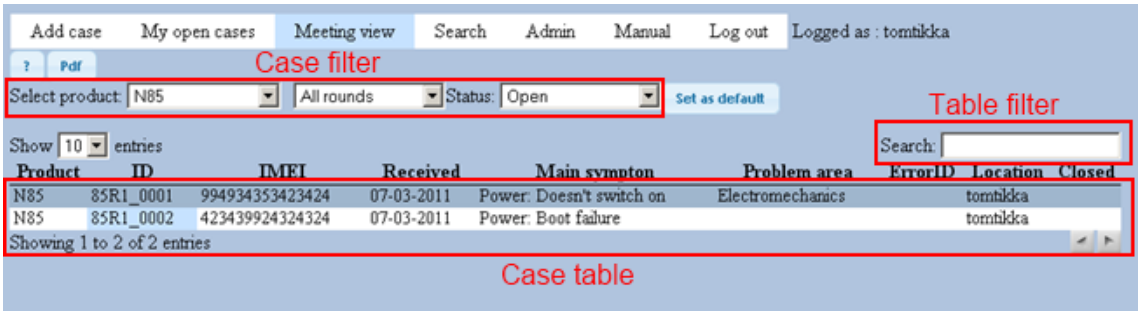

Figure 5.2 Meeting view.

Search view (Figure 5.3) is used to create complex searches to the database. *Search form* provides the possibility of creating a database query without the user having to learn SQL. *Result selection* is used to select how the results are displayed (see section 5.9 for description of the different formats).

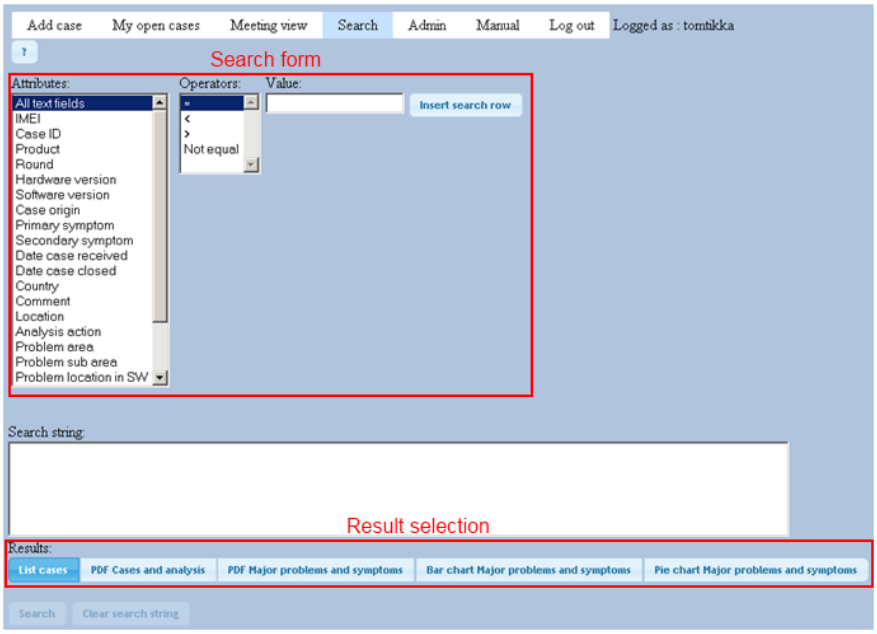

Figure 5.3 Search view.

Admin view (Figure 5.4) is used to maintain the application. *Management console* uses the DataTable to display the managed items. The table can be filtered and sorted, which improves the usability of the console. The view also contains links to additional management views.

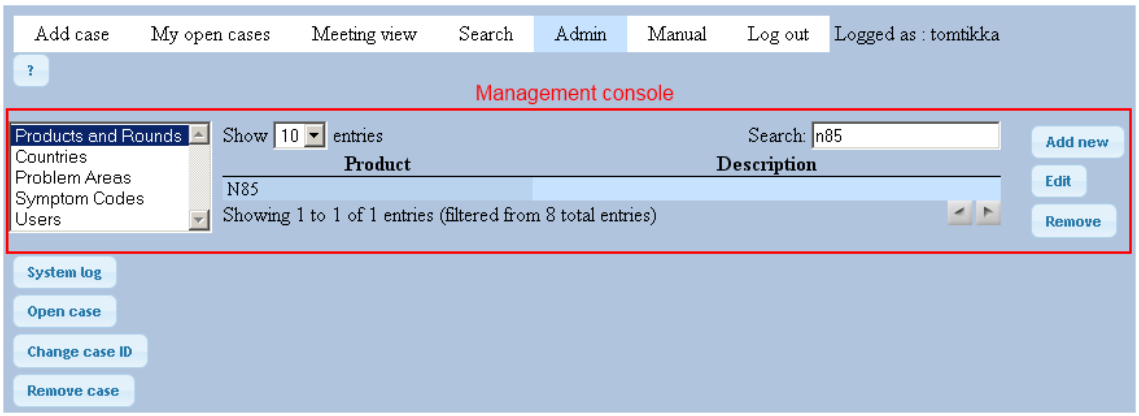

Figure 5.4 Admin view.

5.3 View creation

Controller module classes create the views of the application. A single view is divided to three different parts each represented by a separate class. The topmost part of the page is the template page, which contains the structure and the general layout of a normal view in the application. The template draws all the elements common to all the pages of the application, like the main menu. The template can be further divided to header section and the body section, which are also separate classes. The header section contains the links to the style sheets and libraries used in the page. It also contains any JavaScript code needed in the page. The body section contains the actual content of the page.

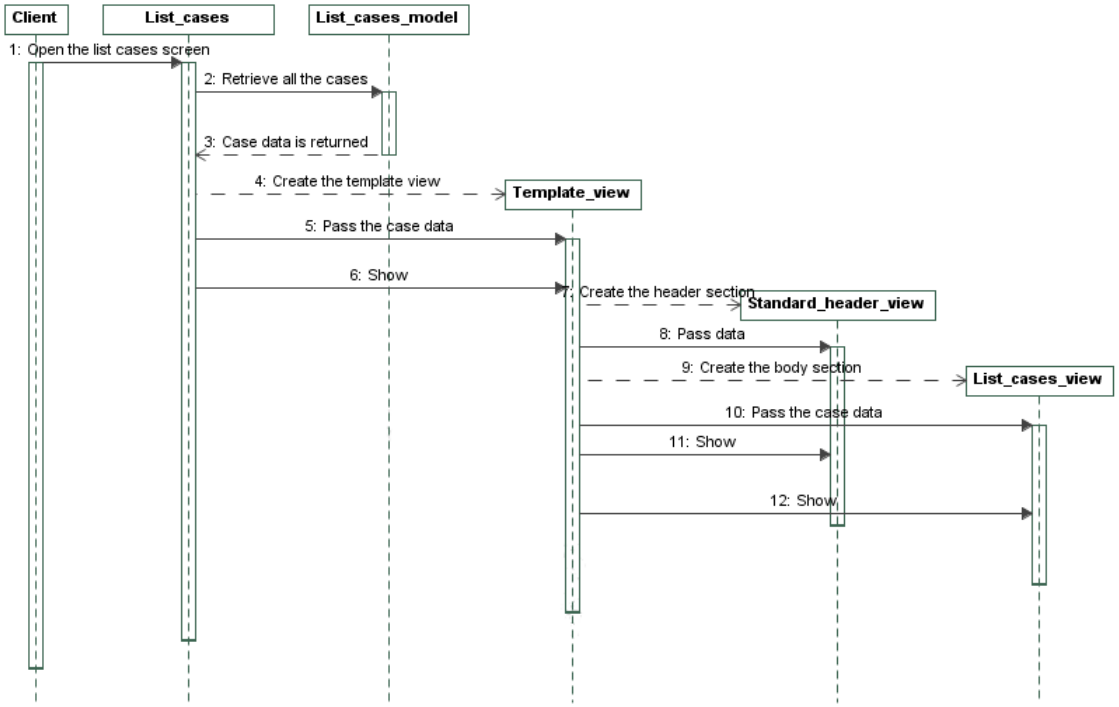

Figure 5.5 The data transfers between modules, when creating a new screen.

Figure 5.5 shows the data transfers between different modules when the user enters a new view in the application. The client browser sends a HTTP request to the server, requesting for the *List cases* view. The CodeIgniter framework directs this request to the controller, which has the same name as the requested resource. The *List_cases* controller then requests all the case data from *List_cases_model*. The model retrieves the data from the database and sends it back to the controller. When the controller receives all the data needed in the *List cases* view, it creates *Template_view* and passes the data to it. After that the controller issues the Show command to *Template_view*, which directs the view to display itself. Before *Template_view* is shown to the user, it first creates *Standard_header_view*, which contains the header section of the HTML page. *Template_view* passes data to it as well. After the header section has been created, *Template_view* creates the body section and passes the case data to it. After all the sections of the *List cases* view have been created, *Template_view* issues the Show command to them, which eventually displays the view to the user.

5.4 Input validation

Most of the data processing is done on the user end in order to reduce the amount of processing required to do in the server end. This processing includes validating data and calculating variety of values. All the user input fields are first checked with JavaScript before being sent to the server for processing. This reduces the load on the server, because it does not have to process illegal data and send the error messages back to the

user. Preventing unnecessary data transfers save bandwidth in both ends and local validation enables to report errors immediately after being discovered.

The processing is done with JavaScript on the user end and with PHP on the server end. When data transfer is required to be performed, it is usually done between the Ajax engine in the client machine and the server. Figure 5.6 shows an example of data transfers between the client and the server in the *Add case* view. The client starts by sending synchronous request to the server requesting for the *Add case* view. The server returns the request HTML page in response.

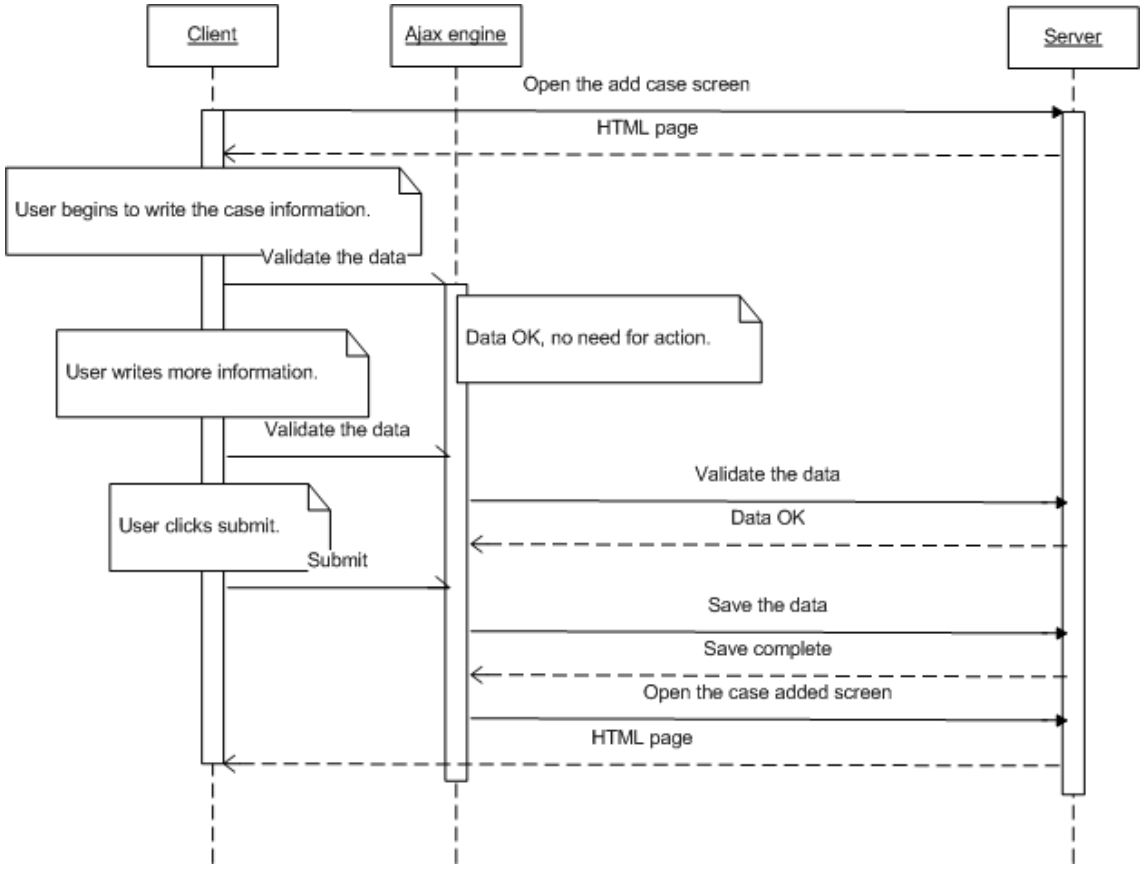

Figure 5.6 Data transfers in the add case screen.

As the user fills the input fields in the new case form, the Ajax engine validates the data as it is being typed. If there is an error, the engine changes the background color of the input field to red and displays an error message. Email field is an example of data, which can be validated locally. The value of the email field can be matched to a regular expression string, which describes the format of the email address. If the value matches the regular expression, it is valid, otherwise it is illegal value and error message is displayed. If the local Ajax engine cannot validate the data, like in a situation where the validation needs a database query to be performed, the engine sends the data to the server in a normal synchronous request. An example of a value, which needs to be checked in the server side, is the IMEI field. The value of the case's IMEI field needs to be unique, no two cases can have the same IMEI number. The field is validated by performing a database search with the given IMEI number. If a case is found with the

same IMEI number, the value is considered to be illegal, otherwise is it valid. This validation process is completely invisible to the user, who only discovers it if an error is displayed indicating illegal data in one of the input fields.

When the user has finished filling the case information and presses the submit button, the Ajax engine revalidates all the input fields. If they are valid, the engine sends a synchronous request to the server requesting for the data to be saved. After the data is saved, the engine redirects the browser to the *Case added* view.

5.5 Data filtering

Like the data validation, the data filtering is done on the user end whenever possible. The application uses the DataTables library extensively, which provides a Microsoft Excel like data table with a filtering feature. The data table contains a search box, which filters the table according to the text passed into it in real time. The filtering is done with JavaScript, so server processing is not required while performing the filtering. The client side data filtering does not reduce the amount of data needed to be send from the server to the client, it only hides the extra data in the user interface. The only way to reduce the amount of data transferred between the server and the client is to perform the data filtering in the server side before the data is transferred.

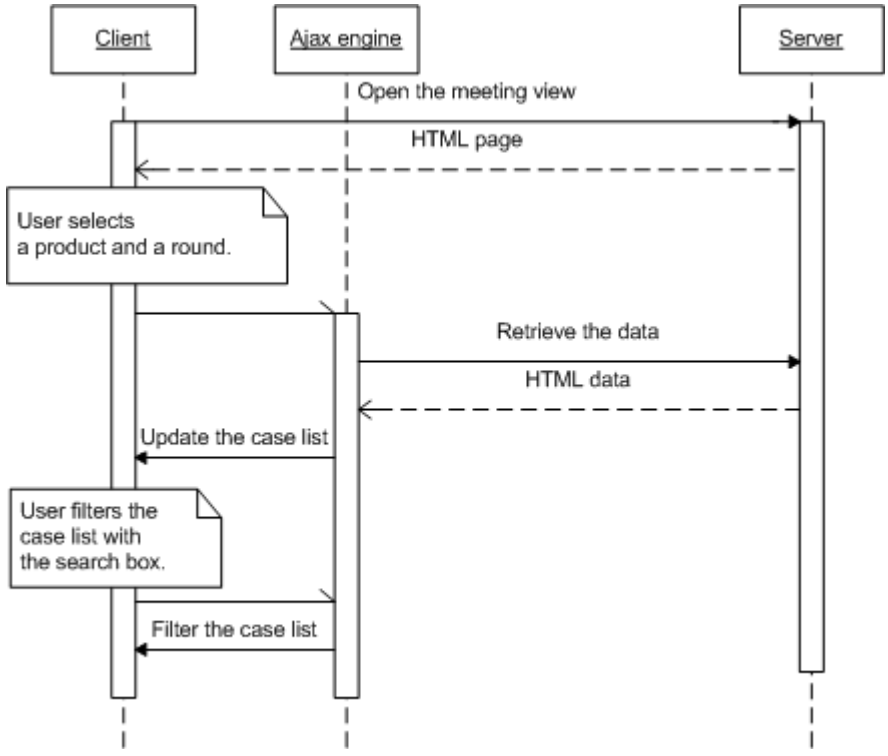

Figure 5.7 Data filtering in the meeting view.

Figure 5.7 shows an example of data filtering being performed in *Meeting view* (Figure 5.5). The process starts with a normal HTTP request and response cycle, where the client requests the *Meeting view* and the server responses by sending the HTML page. The case list in the *Meeting view* can be filtered in the server side with two

dropdown boxes specifying the displayed products and rounds or in the client side with the data table's search box. In the Figure 5.2, the user starts by selecting a product and a round of the cases he/she wants' to see in the case list. After the user has made the selection, the Ajax engine creates a request for the data filtered by the product and round attributes. The user can keep browsing the default case list while the request is being processed. The default list contains all the cases in the database. When the server's response arrives to the client the Ajax engine clear the case list, inserts received data to it and displays it to the user.

After the case list is filtered with the product and round boxes, the user filters it again by typing a filtering text in the data tables search box. This filtering is performed in the Ajax engine, which hides all the cases that do not match the filtering text.

5.6 Case locking

The application is designed for large number of simultaneous users, which means the concurrency issues need to be addressed. If two users edit the same case simultaneously, there is a risk that the user who first updates the case will lose the modifications. The other user does the modifications to the version of the case, which has not been updated according to the modifications done by the first user. This result in the first user losing his modifications after the second user submits updates to the case. To prevent this from happening the application has a locking mechanism for the case editing, which allows only a single user edit a specific case at a time.

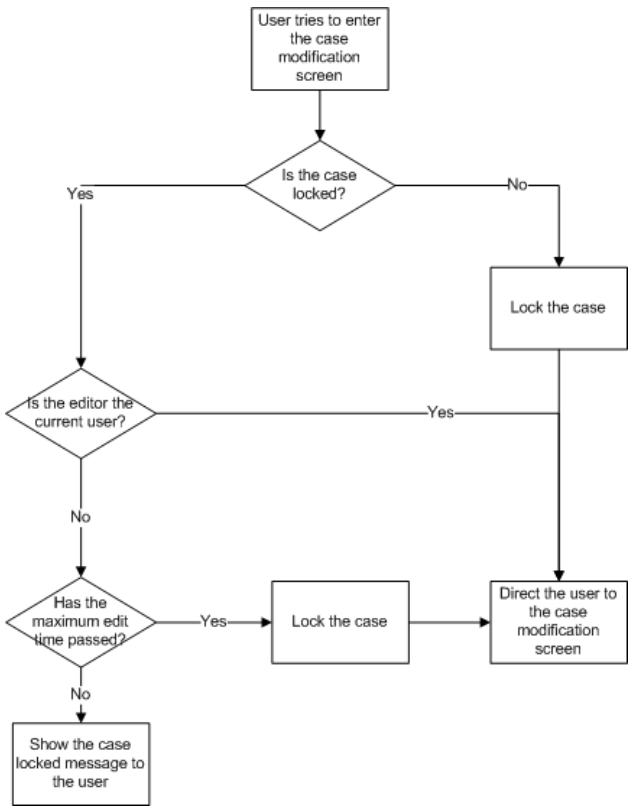

Figure 5.8 The case locking procedure.

The locking is done with the case table having three columns, the case lock status, the user id of the current editor and the time the edit began. Figure 5.8 shows the process of case locking. When a user tries to enter a case modification screen, the lock is first checked to make sure it is open. If it is, then user id column is set according to the current user, the lock is engaged and the user is directed to the *Modify case* screen. If the lock is closed, the user id of the editor is checked if it matches the current user. If it is a match, the user is directed to the case modification screen, because the user is the one who set the case lock in the first place. This situation can occur if the user has opened the edit screen in one browser window and then tries to open it in another. In this situation there is no need to relock the case, because the lock is already in place. The last thing to check if the previous decisions have not resulted in the user getting the case lock, is the edit time that has passed since the case was locked. The case modification screen has a time limit of 25 minutes. If it is exceeded the case lock is opened and the modifications of the user cannot be saved anymore. The purpose of the time limit is to remove the situation, where a user has forgotten to close the edit screen and the case is therefore inaccessible for other users. The whole locking process is done inside a database transaction with the isolation level set to REPEATABLE READ. This prevents multiple users from entering the locking procedure simultaneously.

5.7 Case lifecycle

Figure 5.9 shows the typical lifecycle of a case in the application. It is the responsibility of a person with *Key user* (see Table 5.1) access level to add new cases to the system. The user adds the basic information of the phone that is needed in the analysis. After the phone is added to the application, it is forwarded to an analyzer. As the analysis progresses, the analyzer adds the findings to the case. If the analysis requires some specific knowledge, the analyzer may be changed during the analysis. The new analyzer reads the analysis history of the case and then continues the analysis.

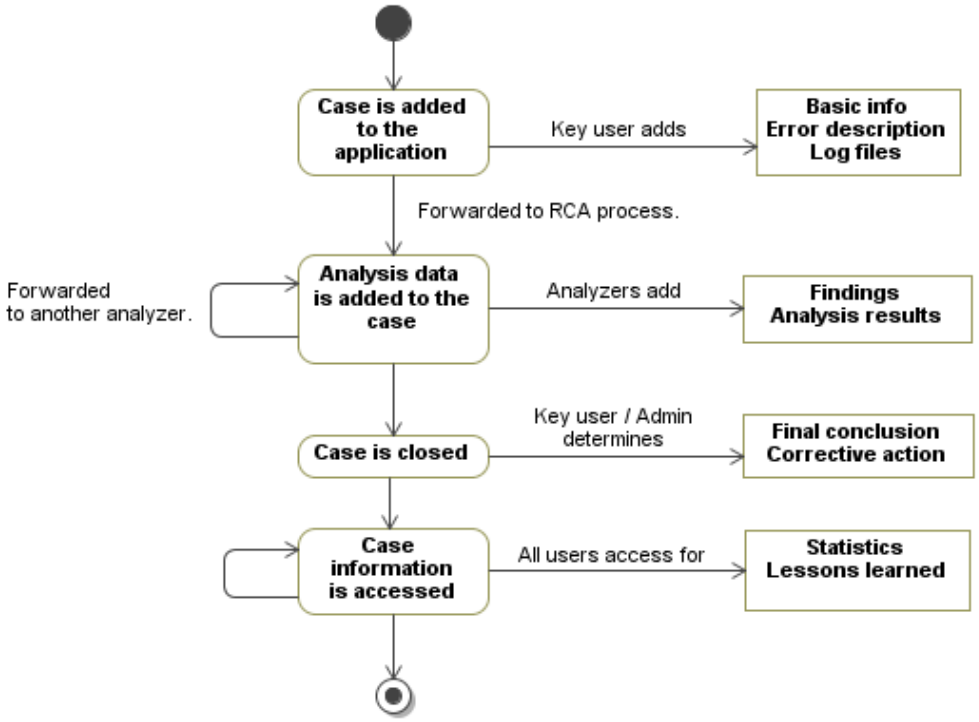

Figure 5.9 A typical lifecycle of a case.

After the root cause is located, the analyzer forwards the case to person, who makes the decision regarding the case. This person then draws the conclusions from the analysis and determines if there is need for any corrective action. After that the case is closed to prevent anyone else modifying it. The information of a closed case can still be read. This is enabled so the statistics may be gathered from a long timeframe and the lessons may be learned from previous analysis. *Admin* (see Table 5.1) can reopen a closed case, but this feature is only for the rare occasions that a case is closed by accident. After the case is closed it stays in the application for as long as the application is in use. *Admin* can remove a case from the application, but like the reopen feature, this is only used when a case has been created by accident.

5.8 Security

The application uses an external enterprise information system for user authentication. Every user who has access to the company intranet has automatically rights to access the application as well. The security of the application is improved with external authentication because there is no need for the user to create additional credentials for the application. Additional credentials might be forgotten or, if written to a paper, end up outside the organization. When a new user logs in the application for the first time, his/her information is retrieved from the external information system and saved to the application's database. This information is limited to the username, user's first and last name and the email address. After the first login, there is no need to retrieve the user's information again. Only the credentials are checked against the external system, which speeds up the login procedure. The application also provides a feature for remembering

the user login. When it is enabled, a cookie is created with an expiration time and an alphanumeric code consisting of 15 letters. The cookie is saved in the user's computer and the code is stored in the database. When a user with a cookie enters the applications login screen, the cookie's expiration time is checked and the code is compared to the code in the database. If they match, the user is automatically set as logged in.

The application has four user access levels (Table 5.1), which are *Blocked user*, *Basic user*, *Key user* and *Admin*. *Blocked user* is a special access level, which prevents the user from logging in to the application. It is used only in the event that there is a malicious user, which must be prohibited from accessing the application. *Basic user* is the default access level set to the new users on their first login to the application. *Key user* role is meant for person, who manages the cases. This includes adding new cases to the system and closing finished cases. An analyzer normally has either *Basic user* or *Key user* rights, depending on the requirements of the work. *Admin* access level is meant for person responsible for maintaining the application.

Table 5.1 Access levels and rights.

Right	*Blocked user*	*Basic user*	*Key user*	*Admin*
Add case	No	No	Yes	Yes
Close case	No	No	Yes	Yes
View case	No	Yes	Yes	Yes
Edit case	No	Yes	Yes	Yes
Remove case	No	No	No	Yes
Re-open case	No	No	No	Yes
Manage support data	No	No	No	Yes

All the modifications the users perform on the cases are recorded to the system log. This information is limited to the username, case id, date and time of the event and to a brief text describing the nature of the modification, like for example "a new case was added to the system". This information can be used to track the modifications and to discover malicious users.

All the data transfer between different HTML pages is filtered to prevent cross-site scripting vulnerabilities. This risk is mostly evident in user input fields, like in the case comments box. If the user injects a malicious JavaScript code in the comments, it is run every time a user views the case. The CodeIgniter framework provides a function for filtering data to prevent XSS (cross-site scripting) vulnerability. This function searches the data for different kinds of JavaScript tags and escapes them. The escaped tags are not recognized as JavaScript tags and the code will not be run. The filtering increases

the server's load slightly and it prevents the inclusions of any kind of JavaScript code in the user input.

5.9 Search and statistics features

The application provides two different ways to search for cases. The first way is provided in the views, which contain *case table* (see section 5.5). The table contains *table filter*, which compares the search string with the contents of the table. This search is run in real time and the table is automatically filtered according to the search string. *Table filter* may be provided with multiple search parameters, which are combined with the logical AND operation. If a more complex search is required, it can be constructed in the *Search* view (Figure 5.7). The view contains a form for creating a complex search string, which is then transformed into a SQL-string. The formed SQL-string is shown to the user and it can be modified directly. The form helps users create complex searches without the need to understand SQL. The search can be selected to display the information of the cases matching the search in *case table* or in a *PDF-report*. There is also an option to display statistics of the cases matching the search. The statistics include the information about major general and detailed problem areas as well as major main and secondary symptoms. These statistics can be shown in either as a PDF-report or as a bar or pie chart.

6 RESULTS

The result of the work done in this thesis is measured in how the application fulfils the requirements set to it.

6.1 Database

6.1.1 Database distribution

When the project was at the planning stage and the milestones were given dates, the requirement for the database distribution (see section 3.3.1) was seen as too large feature to be just a single milestone. So the feature was divided to smaller parts, according to the database partitions, and the contents of the each part were planned in detail. At this stage it was discovered that the information the database needed to store was not fully known. There were plans for new products to be included in the root cause analysis process in the near future, but the information requirements of the devices was not yet known. There were also information requirements from teams that did not participate in the root cause analysis process directly, but which could benefit from the system, if the information that they used could be stored in the database. Gathering all these requirements would have taken much time and the start of the implementation stage would have delayed too long for the project to be finished in four months. Moreover, if the requirements would have changed after the database was completed, the modifications would have been required to be made in three separate database partitions. It was decided for these reasons to drop the requirement for the three separate database partitions. There would be only one single database that all the sites would use. This enabled a more iterate database building process where the database could be modified after it was already in use, because the modifications were needed to made in only one database. This also enabled the implementation and the requirements gathering to proceed simultaneously, which gave more time to gather accurate requirements from multiple end user teams. The drawback of the single database is the efficiency of the database queries. When a user creates a search query for the site's local data, the search is run against the database that holds every site's data. Comparing the search string with other sites data create extra load on the server and with more sites adopting the application, the server's computing requirements grow correspondingly.

6.1.2 Data migration

The data migration from the existing application to the new one (see section 3.2.2) was cancelled due to conflicting data models between the database in the existing application and the database in the new application. The models had different number of columns in the same tables and some of the columns had different lengths and data types. The migration would have required transforming the data significantly and thereby increasing the risk of lowering the integrity of the data. For example, some of the tables in the new database had more columns than the same tables had in the existing database. This means that when the data is migrated to the new database, the values of the new columns need to be filled with default or guessed values. The newly generated values may be incorrect or the new columns may contain information that does not exist in the old data. These risks lower the integrity of the data and thereby diminish the benefits gained from using the application.

6.2 Feature

6.2.1 Case locking

The application has a locking mechanism, which prevents multiple users editing the same case simultaneously. The locking is done in the database and inside a transaction, which makes it an atomic operation. This prevents a race condition in the locking phase and ensures that only single user may edit a particular case at a time, even with huge number of concurrent users. The requirement for case locking (see section 3.3.1) can thus be considered completed.

6.2.2 Messaging tool

The messaging tool feature (see section 3.3.2) was dropped due to lack of interest shown by the end users. An analyst usually has just a few cases open simultaneously, so the problem of forgetting a case is not very common. The feature of notifying the analyzer every time a case is edited by someone else was also met with resistance. There may be cases with many analyzers working at the same time which might result in lots of emails being sent to the analyzer responsible of the case. The analyzer probably already knows that the case is being edited by other analyzers, so the notifications do not provide any new information. The ability to communicate to other analyzers with the messaging tool was also considered to be unnecessary. The organization has already tools for communicating between colleagues, which are better than the messaging tool can be made in the limited time available. Also the users would have to learn to use yet another tool for communication, which would be in this case unnecessary work.

6.2.3 Attachments

The requirement for the metadata to be stored in the database (see section 3.3.3) came in to question after the database distribution requirement was dropped. A central database holding the metadata of multiple sites would grow in size rapidly and increase the storage requirements of the database. The database would also need to be backed up regularly, which creates additional storage requirements and costs. One server holding multiple sites' metadata means that if a user is accessing site's local metadata, like a log file of a device, the client browser will initiate a HTTP connection to the central database server, which may locate over a great distance. This can create a significant latency and if the metadata size is significant, retrieving the data can take considerable time. For these reasons, it was decided that the metadata would not be stored in the database. The metadata was replaced with URL links, which link to the actual location of the metadata. Every site using the application can store the metadata at a location best suited for them, the database contains only a link to the location. This means that accessing local metadata does not require the data being downloaded from a distant server, but it can be retrieved from a local network location. The drawback of linking the metadata is that different sites do not generally have access to each other's local network locations. The users from other sites can see the links to the metadata, but cannot access them.

6.2.4 History data

All the modifications done to the cases are recorded in the database. They can be used to track the changes and to create a timetable of the modifications. For these reasons the requirement for history data (see section 3.3.4) is completed.

6.2.5 URL links

Relevant data can be linked to cases. This completes the requirement for URL links (see section 3.3.5).

6.3 General

6.3.1 Web application

The system is a web-based application. It can be used with web browser and it requires no additional software to be installed. The requirement for web based application (see section 3.4.1) is therefore completed.

6.3.2 Standardize RCA reporting

Another main requirement to the application is to standardize the process of reporting the analysis findings (see section 3.4.2). The application has standard fields for a case's

analysis information. The fields, which contain vital analysis information that must be filled in every case, are mandatory and the case cannot be closed without filling the fields first. This enforces to standardize the analysis information by making sure that every case has the analysis information in the same format and containing the required information. Therefore the requirement has been completed.

6.3.3 Scale to multisite environment

One of the main requirements to the application was that it supports multisite usage (see section 3.4.3), meaning many concurrent users using the application at different times and from different locations. It was planned that the adequate scalability could be gained by using a distributed database. The database distribution feature was later dropped, which leaves the scalability of the application unclear. Some indication of the scalability can be gained by observing the performance of the application currently with little over hundred users and data from the period of six months in the database. Table 6.1 shows the average loading times of the four slowest loading views in the application. The loading time is an average of five separate page loads, with the browser cache cleared after each page load. The loading time represents the time a normal user needs to wait to load the page. The data column displays the amount of data transmitted from the server to the client browser in each page load. This data includes the case related data as well as all the data needed to draw the user interface, like all the CSS and JavaScript files.

Table 6.1 Average loading times of the four slowest loading views.

View	***Average loading time (seconds)***	***Data (KB)***	***Additional information***
Search results	19,70	437	Used full text search.
System log	3,09	1011	
Meeting view	1,37	684	No filter applied.
Case details	0,48	357	

The application loads every view, except the four views in the Table 6.1, so fast that the user does not detect any loading period. The slowest page loading in the application happens, when the user uses *full text search*. This search feature matches the search string to every text field in every table except *User* table. The search uses the SQL's LIKE keyword with the search string surrounded by wildcards that represent any number of any characters. This means that the database system cannot use any indexes to speed up the query and it must scan completely every text field in every table. It is likely that the search time grows linearly with the amount of data in the database. The *System log* view takes on average about three seconds to load with several thousand items in the log. This time is mostly spent on creating the data table with the JavaScript. The view has a feature, which can be used to delete old log files, so the long loading

time of system log is not really an issue. *Meeting view* contains filters which can be used to limit the cases fetched from the database. If the data amounts in the database grow large, the filters can be used to keep *Meeting view* loading times reasonable. The *Case details* view fetches a constant amount of data from the database and indexes are used in the queries the screen generates. It is likely that the loading time of the view stays reasonable even with large data amounts. Overall, the current data amounts are too small as well as the number of users to assess the scalability of the application in multisite environment. For this reason, the requirement is left unresolved.

6.4 Usability

6.4.1 Filtering and search

The application supports searching the data from all the other database tables except the *User* table. The search can include the values of each column of those tables. The search string can be as complex as a normal SQL SELECT query, meaning the application places no limits to what kind of searches the user wants to create. The *Search* view contains a form, which can be used to create complex queries without the need to learn SQL.

All the views containing a data table can be filtered in real time. The data can be filtered using multiple different search parameters, which are combined with the logical AND operation. The filter searches the string from all the columns and all the rows in the data table and hides all the rows, which do not match the filter string. For these reasons the requirement for extensive filtering and searching capabilities (see section 3.5.1) is completed.

6.4.2 User help

All the views of the application contain a help feature. Some of the more specific functions also contain an additional help feature. The main menu contains a link to the user manual. It is a PDF-document, which can be searched with the search function in the PDF-viewer. The requirement for user help functions (see section 3.5.2) is thus completed.

6.5 Overall

Table 6.2 shows how the finished system fulfills the requirements. The status can be *Requirement dropped*, *Unresolved*, *Not completed* or *Completed*.

Table 6.2 Status of the requirements.

Requirement	***Status***
Database distribution	Requirement dropped
Data migration	Requirement dropped
Case locking	Completed
Messaging tool	Requirement dropped
Attachments	Requirement dropped
History data	Completed
URL links	Completed
Web application	Completed
Standardize RCA reporting	Completed
Scale to multisite environment	Unresolved
Filtering and search	Completed
User help	Completed

Many requirements were dropped during the development. The design of the system changed a few times during the development, which made certain requirements obsolete. Overall, the final system fulfills the active requirements well.

7 IDEAS FOR FURTHER DEVELOPMENT

During the development several new ideas emerged, which might improve the scalability, usability and security of the application. Of all these ideas, the site based categorizing is a major enhancement to the application, improving the scalability and security by adding the site information to cases and enabling to filter the data based on this information.

7.1 Categorizing data by sites

Categorizing data by different sites enables more efficient data retrieval and increases the security of the application. Currently the application has no means of searching site specific data, which means that when a user is searching for data concerning a local case, the application compares the search string against the data of all the sites. This creates unnecessary load on the server and increases the time it takes to complete the operation. By adding the site information to the cases, the search area can be limited to the local site and thereby increasing the efficiency of the search functions. The site information can also be used to gather additional data and statistics of the root cause analysis processes. For example, it would be possible to compare the average completion times of RCA processes in different sites or to compare the number of cases or analysts in different sites.

The site information can also be used to increase the security of the application and the integrity of the data in the database. Currently it is possible for any user having higher access level than blocked, to modify any cases in any sites. There is a risk that a user from one site modifies a case in another site either by an accident or maliciously. This risk is increased as the number of sites adopting the application grows and the number of users increases. The site information can be used to limit the editing capabilities of the users to the cases local to their current site. It can also be used to control the process of granting rights to the users. It is now possible for an admin to modify any user's access levels. This creates similar risks as the ability to edit any case in the database. With the site information, the access level granting rights of an admin can be limited to the local site only. This presumes that there is still at least one user, who can set a user in another site as an administrator. The site-based categorizing can be made by creating a new table in the database, which holds the site data. The tables holding the case and user information are then linked with the site table, thereby creating a connection between a case and the site it is located as well as a user and his/her local site.

7.2 Filtering by multiple products

All the views containing *case table* should have a filtering option, which can be used to select multiple different products and rounds shown on the table. This would enable the user to examine simultaneously multiple cases from different products without the need to create a matching search query in the *Search* view. It is common usage scenario, where the analyst needs to simultaneously examine multiple cases, which might belong to different products. Currently the views with *case table* contain *table filter*, which can be used to filter the table to a specific product, but not to multiple specific products. The only way to filter multiple products is to create a matching search string in the *Search* view, which takes time and requires the user to leave the view with the *case table*.

7.3 Automatic backups

Automatic database backup tool could help make the application more maintainable. The application's database contains vital information, which needs regularly backups. Currently the application has no backup function, the database backups need to be made manually outside the application. This creates additional work to the person in charge of maintaining the application and it makes it more difficult to deploy the application to a new site. The backup tool should operate automatically at user defined times, so minimal amount of user interaction is needed and the backups can be made at times the database server is under minimum load. The backup destination should also be user definable, so the backups can be placed in separate physical drives, if needed.

7.4 Group-based access policy

Currently the system has four user access levels with predefined set of rights to the application (see section 5.8). This is sufficient for the current usage, but there may rise additional requirements in the future. For example, there may rise the need for more user levels or the need to set the rights of each user individually. This can be accomplished with group-based access policy, where the access levels are set to groups and the users are then set as members of those groups. A single user may belong to a single group or to many groups depending on the work requirements. The group-based access policy enables to create easily the access level profile of the standard users. The needed standard rights can be given to a single group, which is then set as the default group of a new user. If there is a need for a user with special privileges, a new group can be created with those privileges and the user can then be set as a member of that group.

The group-based access policy requires that four new tables are created to the database. One table is for the group, one for the access right and one table to join the previous two tables. The last table is used to link the user table and the group table.

7.5 System log backup

The system log contains a brief description of all the modifications done to the cases. The size of the log can grow very fast when many users are doing modifications to the cases. With thousands of log notes, the loading of the *System log* view can take seconds and use megabytes of memory in the client machine. The *System log* view has a delete feature, which can be used to remove old log notes, but there is no method, which could be used to copy the notes to another location before removing them. An export feature should be added to the *System log* view for exporting the log notes to a text or PDF file. The file can then be saved to the client machine. This would improve the data security by providing a way to backup the system log. It would also enable the administrator to keep the majority of the log notes in different location and to keep only the latest notes in the database.

8 CONCLUSIONS

Root cause analysis is an important part of the mobile phone product development process. It provides valuable information about a failure, which can be used to improve the quality of the devices. The recommendations provided by the analysis help to avoid the failure in the future products and may offer a way to fix devices already in the market. The aim of this thesis was to design and implement a reporting system to the analysis process. The system provides a single point where all the analysis data may be stored and it standardizes the process of reporting the findings. This improves the visibility of the process and its findings to other sites and R&D programmes. Considering the requirements, the resulting application fulfills them well.

The application designed in this thesis contains a database and a web user interface for easy access to the data. The application can be used without any additional software and every user with intranet credentials to the organization can login to the application. These features make it easy to spread the information provided by the process across the organization. By providing a central location to all of the analysis data, the data can be maintained more easily, it is always up to date and the data modifications can be controlled with multiple access levels. The mandatory fields for the analysis data standardize the process of reporting the findings and enable to compare the findings from multiple different sites.

All but one requirement can be considered completed. The requirement for the scalability of the application to multisite usage remains still unclear. Currently the application has been in beta test for six months with little over hundred users using the application. The data amounts are still too low to assess how the application performs in multisite environment with many concurrent users and significant data amounts in the database.

Overall the project went quite smoothly, which is mainly due to active end user involvement. A small group of the end users of the application was involved with the design of the application from the start. They provided continuous feedback on the features and the direction the application was taken. This enabled a more iterative development cycle, which improved particularly the usability of the finished system.

As there are currently over hundred active users using the application in their daily work and the user feedback have been positive, the work done in this thesis can be considered a success. The application contains features, which improve the root cause analysis process and thereby the quality of mobile phones. The application however lacks some features, which might further benefit the organization using the application.

Among these, a site-based data categorizing stands as the one feature, which should be considered if the application is to be used in multisite environment.

There are plans to take the application to production use within the near future. Before that it needs more testing to make sure the integrity of the database stays consistent with more than hundred users and the application responds to the user input within reasonable amount of time.

REFERENCES

Andersen, B., Fagerhaug, T. 2006. Root cause analysis: simplified tools and techniques. Milwaukee, ASQ Quality Press. 240 p.

Asleson, R., Schutta, N.T. 2006. Ajax – Tehokas hallinta. Jyväskylä, Gummerus Kirjapaino Oy. 273 p.

Base, R. 2008. Implementing Six Sigma and Lean: a practical guide to tools and techniques. Jordan Hill, Oxford, Elsevier Linacre House. 342 p.

Bergman, B.L.S, Fundin, A.P., Gremyr, I.C, Johansson, P.M. Beyond root cause analysis. IEEE Reliability and maintainability symposium. 2002. pp. 140-146.

Berners-Lee, T., Fielding, R., Irvine, U.C., Masinter, L. Uniform Resource Identifiers (URI): Generic Syntax. Network Working Group RFC 2616. 1998. [accessed 23.1.2011]. Available: http://www.ietf.org/rfc/rfc2396.txt.

Cox, J.F., Spencer, M.S. 1998. The constraints management handbook. Boca Raton, Florida, St. Lucie Press. 352 p.

Doggett, A.M. 2004. A statistical comparison of three root cause analysis tools. Journal of industrial technology 20, 2, pp. 1-9.

Doggett, A.M. 2005. Root cause analysis: a framework for tool selection. The Quality Management Journal, 2005. Vol. 12. No. 4. ABI/INFORM Global, pp. 34-45.

Fielding, R., Irvine, U.C., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T. Hypertext Transfer Protocol -- HTTP/1.1. Internet Engineering Task Force RFC 2616. 1999. [accessed 23.1.2011]. Available: http://www.ietf.org/rfc/rfc2616.txt.

Garrett, J.J. 2005. Ajax: A New Approach to Web Applications. [www]. [accessed 4.2.2011]. Available: http://www.adaptivepath.com/publications/essays/archives/000385.php

Ishikawa, K. 1982. Guide to quality control, second edition. Tokyo: Asian Productivity Organization. 226 p.

IEEE Power Engineering Society. IEEE Guide for Induction Machinery Maintenance Testing and Failure Analysis. IEEE Std 1415-2006, 2007. pp. 1-58.

Julisch, K. Using Root Cause Analysis to Handle Intrusion Detection Alarms. Dissertation. Dortmund 2003. University of Dortmund. p. 137.

Mabin, V.J., Forgeson, S., Green, L. 2001. Harnessing resistance: using the theory of constraints to assist change management. Journal of European Industrial Training, 25, 2/3/4, pp. 168-191.

Milosevic, D.Z. 2003. Project management toolbox: tools and techniques for the practising project manager. Hoboken, New Jersey, John Wiley & Sons, Inc. 600 p.

Mobley, R.K. 1999. Root cause failure analysis. Elsevier. [accessed 30.12.2010]. Available: http://knovel.com/web/portal/browse/display?_EXT_KNOVEL_DISPLAY_bookid=443&VerticalID=0.

Mogul, J.C. The Case for Persistent-Connection HTTP. ACM SIGCOMM Computer Communication Review 25(1995)4, pp. 299-313.

Paradies, M., Busch, D. Root cause analysis at Savannah River plant [nuclear power station]. IEEE Human factors and power plants conference. 1988. pp. 479-483. [accessed 6.1.2011]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=27547&isnumber=1061. Limited availability.

Paulson, L.D. Building rich web applications with Ajax. Computer, 38(2005)10, pp. 14–17.

Rooney, J.J., Vanden Heuvel, L.N. Quality basics: Root cause analysis for beginners. Quality Progress (2004)37, pp. 45-53.

Shklar, L., Rosen, R. 2003. Web Application Architecture: principles, protocols and practices. Chichester, West Sussex, John Wiley & Sons, Ltd. 374 p.

Thiemann, P. WASH/CGI: Server-side Web Scripting with Sessions and Typed, Compositional Forms. Practical Aspects of Declarative Languages, 2002.

Trent, S., Tatsubori, M., Suzumara, T., Tozawa, A., Onodera, T. Performance Comparison of PHP and JSP as Server-Side Scripting Languages. Proceedings of Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008. Springer, 2008. pp. 164–182.

Wilson, P.F., Dell, L.D., & Anderson, G.F. 1993. Root Cause Analysis, A Tool For Total Quality Management, ASQC Quality Press. 216 p.